Chapter 3

Core Java APIs

OCA EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Using Operators and Decision Constructs**

- Test equality between Strings and other objects using == and equals()

✓ **Creating and Using Arrays**

- Declare, instantiate, initialize and use a one-dimensional array
- Declare, instantiate, initialize and use a multi-dimensional array

✓ **Working with Selected classes from the Java API**

- Creating and manipulating Strings
- Manipulate data using the StringBuilder class and its methods
- Declare and use an ArrayList of a given type
- Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period

✓ **Working with Java Data Types**

- Develop code that uses wrapper classes such as Boolean, Double, and Integer.

The OCA exam expects you to know the core data structures and classes used in Java, and in this chapter readers will learn about the most common methods available. For example, `String` and `StringBuilder` are used for text data. An array and an `ArrayList` are used when you have multiple values. A variety of classes are used for working with dates. In this chapter, you'll also learn how to determine whether two objects are equal.

API stands for application programming interface. In Java, an interface is something special. In the context of an API, it can be a group of class or interface definitions that gives you access to a service or functionality. You will learn about the most common APIs for each of the classes covered in this chapter.

Creating and Manipulating Strings

The `String` class is such a fundamental class that you'd be hard-pressed to write code without it. After all, you can't even write a `main()` method without using the `String` class. A *string* is basically a sequence of characters; here's an example:

```
String name = "Fluffy";
```

As you learned in Chapter 1, "Java Building Blocks," this is an example of a reference type. You also learned that reference types are created using the `new` keyword. Wait a minute. Something is missing from the previous example: it doesn't have `new` in it! In Java, these two snippets both create a `String`:

```
String name = "Fluffy";
String name = new String("Fluffy");
```

Both give you a reference variable of type *name* pointing to the `String` object `"Fluffy"`. They are subtly different, as you'll see in the section "String Pool," later in this chapter. For now, just remember that the `String` class is special and doesn't need to be instantiated with `new`.

In this section, we'll look at concatenation, immutability, the string pool, common methods, and method chaining.

Concatenation

In Chapter 2, "Operators and Statements," you learned how to add numbers. 1 + 2 is clearly 3. But what is `"1" + "2"`? It's actually `"12"` because Java combines the two `String` objects.

Placing one `String` before the other `String` and combining them together is called string *concatenation*. The OCA exam creators like string concatenation because the `+` operator can be used in two ways within the same line of code. There aren't a lot of rules to know for this, but you have to know them well:

1. If both operands are numeric, `+` means numeric addition.
2. If either operand is a `String`, `+` means concatenation.
3. The expression is evaluated left to right.

Now let's look at some examples:

```
System.out.println(1 + 2);           // 3
System.out.println("a" + "b");       // ab
System.out.println("a" + "b" + 3);   // ab3
System.out.println(1 + 2 + "c");     // 3c
```

The first example uses the first rule. Both operands are numbers, so we use normal addition. The second example is simple string concatenation, described in the second rule. The quotes for the `String` are only used in code—they don't get output.

The third example combines both the second and third rules. Since we start on the left, Java figures out what `"a" + "b"` evaluates to. You already know that one: it's `"ab"`. Then Java looks at the remaining expression of `"ab" + 3`. The second rule tells us to concatenate since one of the operands is a `String`.

In the fourth example, we start with the third rule, which tells us to consider `1 + 2`. Both operands are numeric, so the first rule tells us the answer is `3`. Then we have `3 + "c"`, which uses the second rule to give us `"3c"`. Notice all three rules get used in one line? The exam takes this a step further and will try to trick you with something like this:

```
int three = 3;
String four = "4";
System.out.println(1 + 2 + three + four);
```

When you see this, just take it slow and remember the three rules—and be sure to check the variable types. In this example, we start with the third rule, which tells us to consider `1 + 2`. The first rule gives us `3`. Next we have `3 + three`. Since `three` is of type `int`, we still use the first rule, giving us `6`. Next we have `6 + four`. Since `four` is of type `String`, we switch to the second rule and get a final answer of `"64"`. When you see questions like this, just take your time and check the types. Being methodical pays off.

There is only one more thing to know about concatenation, but it is an easy one. In this example, you just have to remember what `+=` does. `s += "2"` means the same thing as `s = s + "2"`.

```
4: String s = "1";             // s currently holds "1"
5: s += "2";                   // s currently holds "12"
6: s += 3;                     // s currently holds "123"
7: System.out.println(s);      // 123
```

On line 5, we are "adding" two strings, which means we concatenate them. Line 6 tries to trick you by adding a number, but it's just like we wrote `s = s + 3`. We know that a string "plus" anything else means to use concatenation.

To review the rules one more time: use numeric addition if two numbers are involved, use concatenation otherwise, and evaluate from left to right. Have you memorized these three rules yet? Be sure to do so before the exam!

Immutability

Once a `String` object is created, it is not allowed to change. It cannot be made larger or smaller, and you cannot change one of the characters inside it.

You can think of a string as a storage box you have perfectly full and whose sides can't bulge. There's no way to add objects, nor can you replace objects without disturbing the entire arrangement. The trade-off for the optimal packing is zero flexibility.

Mutable is another word for changeable. *Immutable* is the opposite—an object that can't be changed once it's created. On the OCA exam, you need to know that `String` is immutable.

More on Immutability

You won't be asked to identify whether custom classes are immutable on the exam, but it's helpful to see an example. Consider the following code:

```
class Mutable {
  private String s;
  public void setS(String newS){ s = newS; }  // Setter makes it mutable
  public String getS() { return s; }
}
final class Immutable {
  private String s = "name";
  public String getS() { return s; }
}
```

`Immutable` only has a getter. There's no way to change the value of *s* once it's set. `Mutable` has a setter as well. This allows the reference *s* to change to point to a different `String` later. Note that even though the `String` class is immutable, it can still be used in a mutable class. You can even make the instance variable `final` so the compiler reminds you if you accidentally change *s*.

Also, immutable classes in Java are final, and subclasses can't add mutable behavior.

You learned that + is used to do `String` concatenation in Java. There's another way, which isn't used much on real projects but is great for tricking people on the exam. What does this print out?

```
String s1 = "1";
String s2 = s1.concat("2");
s2.concat("3");
System.out.println(s2);
```

Did you say `"12"`? Good. The trick is to see if you forget that the String class is immutable by throwing a method at you.

The String Pool

Since strings are everywhere in Java, they use up a lot of memory. In some production applications, they can use up 25–40 percent of the memory in the entire program. Java realizes that many strings repeat in the program and solves this issue by reusing common ones. The *string pool*, also known as the intern pool, is a location in the Java virtual machine (JVM) that collects all these strings.

The string pool contains literal values that appear in your program. For example, "name" is a literal and therefore goes into the string pool. `myObject.toString()` is a string but not a literal, so it does not go into the string pool. Strings not in the string pool are garbage collected just like any other object.

Remember back when we said these two lines are subtly different?

```
String name = "Fluffy";
String name = new String("Fluffy");
```

The former says to use the string pool normally. The second says "No, JVM. I really don't want you to use the string pool. Please create a new object for me even though it is less efficient." When you write programs, you wouldn't want to do this. For the exam, you need to know that it is allowed.

Important String Methods

The `String` class has dozens of methods. Luckily, you need to know only a handful for the exam. The exam creators pick most of the methods developers use in the real world.

For all these methods, you need to remember that a string is a sequence of characters and Java counts from 0 when indexed. Figure 3.1 shows how each character in the string `"animals"` is indexed.

FIGURE 3.1 Indexing for a string

a	n	i	m	a	l	s
0	1	2	3	4	5	6

Let's look at thirteen methods from the `String` class. Many of them are straightforward so we won't discuss them at length. You need to know how to use these methods.

length()

The method `length()` returns the number of characters in the `String`. The method signature is as follows:

```
int length()
```

The following code shows how to use `length()`:

```
String string = "animals";
System.out.println(string.length());  // 7
```

Wait. It outputs 7? Didn't we just tell you that Java counts from 0? The difference is that zero counting happens only when you're using indexes or positions within a list. When determining the total size or length, Java uses normal counting again.

charAt()

The method `charAt()` lets you query the string to find out what character is at a specific index. The method signature is as follows:

```
char charAt(int index)
```

The following code shows how to use `charAt()`:

```
String string = "animals";
System.out.println(string.charAt(0));  // a
System.out.println(string.charAt(6));  // s
System.out.println(string.charAt(7));  // throws exception
```

Since indexes start counting with 0, `charAt(0)` returns the "first" character in the sequence. Similarly, `charAt(6)` returns the "seventh" character in the sequence. `charAt(7)` is a problem. It asks for the "eighth" character in the sequence, but there are only seven characters present. When something goes wrong that Java doesn't know how to deal with, it throws an exception, as shown here. You'll learn more about exceptions in Chapter 6, "Exceptions."

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 7
```

indexOf()

The method `indexOf()` looks at the characters in the string and finds the first index that matches the desired value. `indexOf` can work with an individual character or a whole `String` as input. It can also start from a requested position. The method signatures are as follows:

```
int indexOf(char ch)
int indexOf(char ch, index fromIndex)
int indexOf(String str)
int indexOf(String str, index fromIndex)
```

The following code shows how to use `indexOf()`:

```
String string = "animals";
System.out.println(string.indexOf('a'));          // 0
System.out.println(string.indexOf("al"));         // 4
System.out.println(string.indexOf('a', 4));       // 4
System.out.println(string.indexOf("al", 5));      // -1
```

Since indexes begin with 0, the first `'a'` matches at that position. The second statement looks for a more specific string and so matches later on. The third statement says Java shouldn't even look at the characters until it gets to index 4. The final statement doesn't find anything because it starts looking after the match occurred. Unlike `charAt()`, the `indexOf()` method doesn't throw an exception if it can't find a match. `indexOf()` returns –1 when no match is found. Because indexes start with 0, the caller knows that –1 couldn't be a valid index. This makes it a common value for a method to signify to the caller that no match is found.

substring()

The method `substring()` also looks for characters in a string. It returns parts of the string. The first parameter is the index to start with for the returned string. As usual, this is a zero-based index. There is an optional second parameter, which is the end index you want to stop at.

Notice we said "stop at" rather than "include." This means the `endIndex` parameter is allowed to be 1 past the end of the sequence if you want to stop at the end of the sequence. That would be redundant, though, since you could omit the second parameter entirely in that case. In your own code, you want to avoid this redundancy. Don't be surprised if the exam uses it though. The method signatures are as follows:

```
int substring(int beginIndex)
int substring(int beginIndex, int endIndex)
```

The following code shows how to use `substring()`:

```
String string = "animals";
System.out.println(string.substring(3));  // mals
System.out.println(string.substring(string.indexOf('m'))); // mals
System.out.println(string.substring(3, 4)); // m
System.out.println(string.substring(3, 7)); // mals
```

The `substring()` method is the trickiest `String` method on the exam. The first example says to take the characters starting with index 3 through the end, which gives us `"mals"`. The second example does the same thing: it calls `indexOf()` to get the index rather than hard-coding it. This is a common practice when coding because you may not know the index in advance.

The third example says to take the characters starting with index 3 until, but not including, the character at index 4—which is a complicated way of saying we want a `String` with one character: the one at index 3. This results in `"m"`. The final example says to take the characters starting with index 3 until we get to index 7. Since index 7 is the same as the end of the string, it is equivalent to the first example.

We hope that wasn't too confusing. The next examples are less obvious:

```
System.out.println(string.substring(3, 3)); // empty string
System.out.println(string.substring(3, 2));  // throws exception
System.out.println(string.substring(3, 8)); // throws exception
```

The first example in this set prints an empty string. The request is for the characters starting with index 3 until you get to index 3. Since we start and end with the same index, there are *no* characters in between. The second example in this set throws an exception because the indexes can be backward. Java knows perfectly well that it will never get to index 2 if it starts with index 3. The third example says to continue until the eighth character. There is no eighth position, so Java throws an exception. Granted, there is no seventh character either, but at least there is the "end of string" invisible position.

Let's review this one more time since `substring()` is so tricky. The method returns the string starting from the requested index. If an end index is requested, it stops right before that index. Otherwise, it goes to the end of the string.

toLowerCase() and *toUpperCase()*

Whew. After that mental exercise, it is nice to have methods that do exactly what they sound like! These methods make it easy to convert your data. The method signatures are as follows:

```
String toLowerCase(String str)
String toUpperCase(String str)
```

The following code shows how to use these methods:

```
String string = "animals";
System.out.println(string.toUpperCase());  // ANIMALS
System.out.println("Abc123".toLowerCase());  // abc123
```

These methods do what they say. `toUpperCase()` converts any lowercase characters to uppercase in the returned string. `toLowerCase()` converts any uppercase characters to lowercase in the returned string. These methods leave alone any characters other than letters. Also, remember that strings are immutable, so the original string stays the same.

equals() and *equalsIgnoreCase()*

The `equals()` method checks whether two `String` objects contain exactly the same characters in the same order. The `equalsIgnoreCase()` method checks whether two `String` objects contain the same characters with the exception that it will convert the characters' case if needed. The method signatures are as follows:

```
boolean equals(String str)
boolean equalsIgnoreCase(String str)
```

The following code shows how to use these methods:

```
System.out.println("abc".equals("ABC"));  // false
System.out.println("ABC".equals("ABC"));  // true
System.out.println("abc".equalsIgnoreCase("ABC"));  // true
```

This example should be fairly intuitive. In the first example, the values aren't exactly the same. In the second, they are exactly the same. In the third, they differ only by case, but it is okay because we called the method that ignores differences in case.

startsWith() and *endsWith()*

The `startsWith()` and `endsWith()` methods look at whether the provided value matches part of the `String`. The method signatures are as follows:

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

The following code shows how to use these methods:

```
System.out.println("abc".startsWith("a")); // true
System.out.println("abc".startsWith("A")); // false
System.out.println("abc".endsWith("c")); // true
System.out.println("abc".endsWith("a")); // false
```

Again, nothing surprising here. Java is doing a case-sensitive check on the values provided.

contains()

The `contains()` method also looks for matches in the `String`. It isn't as particular as `startsWith()` and `endsWith()`—the match can be anywhere in the `String`. The method signature is as follows:

```
boolean contains(String str)
```

The following code shows how to use these methods:

```
System.out.println("abc".contains("b")); // true
System.out.println("abc".contains("B")); // false
```

Again, we have a case-sensitive search in the `String`. The `contains()` method is a convenience method so you don't have to write **`str.indexOf(otherString) != -1`**.

replace()

The `replace()` method does a simple search and replace on the string. There's a version that takes `char` parameters as well as a version that takes `CharSequence` parameters. A `CharSequence` is a general way of representing several classes, including `String` and `StringBuilder`. It's called an interface, which we'll cover in Chapter 5, "Class Design." The method signatures are as follows:

```
String replace(char oldChar, char newChar)
String replace(CharSequence oldChar, CharSequence newChar)
```

The following code shows how to use these methods:

```
System.out.println("abcabc".replace('a', 'A')); // AbcAbc
System.out.println("abcabc".replace("a", "A")); // AbcAbc
```

The first example uses the first method signature, passing in `char` parameters. The second example uses the second method signature, passing in `String` parameters.

trim()

You've made it through the all the `String` methods you need to know except one. We left the easy one for last. The `trim()` method removes whitespace from the beginning and end of a `String`. In terms of the exam, whitespace consists of spaces along with the `\t` (tab) and `\n` (newline) characters. Other characters, such as `\r` (carriage return), are also included in what gets trimmed. The method signature is as follows:

```
public String trim()
```

The following code shows how to use this method:

```
System.out.println("abc".trim());           // abc
System.out.println("\t   a b c\n".trim()); // a b c
```

The first example prints the original string because there are no whitespace characters at the beginning or end. The second example gets rid of the leading tab, subsequent spaces, and the trailing newline. It leaves the spaces that are in the middle of the string.

Method Chaining

It is common to call multiple methods on the same `String`, as shown here:

```
String start = "AniMaL   ";
String trimmed = start.trim();                 // "AniMaL"
```

```
String lowercase = trimmed.toLowerCase();      // "animal"
String result = lowercase.replace('a', 'A');   // "Animal"
System.out.println(result);
```

This is just a series of `String` methods. Each time one is called, the returned value is put in a new variable. There are four `String` values along the way, and `Animal` is output.

However, on the exam there is a tendency to cram as much code as possible into a small space. You'll see code using a technique called method chaining. Here's an example:

```
String result = "AniMaL   ".trim().toLowerCase().replace('a', 'A');
System.out.println(result);
```

This code is equivalent to the previous example. It also creates four `String` objects and outputs `Animal`. To read code that uses method chaining, start at the left and evaluate the first method. Then call the next method on the returned value of the first method. Keep going until you get to the semicolon.

Remember that `String` is immutable. What do you think the result of this code is?

```
5: String a = "abc";
6: String b = a.toUpperCase();
7: b = b.replace("B", "2").replace('C', '3');
8: System.out.println("a=" + a);
9: System.out.println("b=" + b);
```

On line 5, we set *a* to point to `"abc"` and never pointed *a* to anything else. Since we are dealing with an immutable object, none of the code on lines 6 or 7 changes *a*.

b is a little trickier. Line 6 has *b* pointing to `"ABC"`, which is straightforward. On line 7, we have method chaining. First, "`"ABC".replace("B", "2")` is called. This returns `"A2C"`. Next, `"A2C".replace("'C', '3')` is called. This returns `"A23"`. Finally, *b* changes to point to this returned `String`. When line 9 executes, *b* is `"A23"`.

Using the *StringBuilder* Class

A small program can create a lot of `String` objects very quickly. For example, how many do you think this piece of code creates?

```
10: String alpha = "";
11: for(char current = 'a'; current <= 'z'; current++)
12:  alpha += current;
13: System.out.println(alpha);
```

The empty `String` on line 10 is instantiated, and then line 12 appends an `"a"`. However, because the `String` object is immutable, a new `String` object is assigned to *alpha* and the

"" object becomes eligible for garbage collection. The next time through the loop, *alpha* is assigned a new `String` object, `"ab"`, and the `"a"` object becomes eligible for garbage collection. The next iteration assigns *alpha* to `"abc"` and the `"ab"` object becomes eligible for garbage collection, and so on.

This sequence of events continues, and after 26 iterations through the loop, a total of 27 objects are instantiated, most of which are immediately eligible for garbage collection.

This is very inefficient. Luckily, Java has a solution. The `StringBuilder` class creates a `String` without storing all those interim `String` values. Unlike the `String` class, `StringBuilder` is not immutable.

```
15: StringBuilder alpha = new StringBuilder();
16: for(char current = 'a'; current <= 'z'; current++)
17:  alpha.append(current);
18: System.out.println(alpha);
```

On line 15, a new `StringBuilder` object is instantiated. The call to `append()` on line 17 adds a character to the `StringBuilder` object each time through the for loop and appends the value of current to the end of *alpha*. This code reuses the same `StringBuilder` without creating an interim `String` each time.

In this section, we'll look at creating a `StringBuilder`, common methods, and a comparison to `StringBuffer`.

Mutability and Chaining

We're sure you noticed this from the previous example, but `StringBuilder` is not immutable. In fact, we gave it 27 different values in the example (blank plus adding each letter in the alphabet). The exam will likely try to trick you with respect to `String` and `StringBuilder` being mutable.

Chaining makes this even more interesting. When we chained `String` method calls, the result was a new `String` with the answer. Chaining `StringBuilder` objects doesn't work this way. Instead, the `StringBuilder` changes its own state and returns a reference to itself! Let's look at an example to make this clearer:

```
4: StringBuilder sb = new StringBuilder("start");
5: sb.append("+middle");                    // sb = "start+middle"
6: StringBuilder same = sb.append("+end");  // "start+middle+end"
```

Line 5 adds text to the end of *sb*. It also returns a reference to *sb*, which is ignored. Line 6 also adds text to the end of *sb* and returns a reference to *sb*. This time the reference is stored in *same*—which means *sb* and *same* point to the exact same object and would print out the same value.

The exam won't always make the code easy to read by only having one method per line. What do you think this example prints?

```
4: StringBuilder a = new StringBuilder("abc");
5: StringBuilder b = a.append("de");
```

```
6: b = b.append("f").append("g");
7: System.out.println("a=" + a);
8: System.out.println("b=" + b);
```

Did you say both print `"abcdefg"`? Good. There's only one `StringBuilder` object here. We know that because `new StringBuilder()` was called only once. On line 5, there are two variables referring to that object, which has a value of `"abcde"`. On line 6, those two variables are still referring to that same object, which now has a value of `"abcdefg"`. Incidentally, the assignment back to *b* does absolutely nothing. *b* is already pointing to that `StringBuilder`.

Creating a *StringBuilder*

There are three ways to construct a `StringBuilder`:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```

The first says to create a `StringBuilder` containing an empty sequence of characters and assign *sb1* to point to it. The second says to create a `StringBuilder` containing a specific value and assign *sb2* to point to it. For the first two, it tells Java to manage the implementation details. The final example tells Java that we have some idea of how big the eventual value will be and would like the `StringBuilder` to reserve a certain number of slots for characters.

Size vs. Capacity

The behind-the-scenes process of how objects are stored isn't on the exam, but some knowledge of this process may help you better understand and remember `StringBuilder`.

Size is the number of characters currently in the sequence, and capacity is the number of characters the sequence can currently hold. Since a `String` is immutable, the size and capacity are the same. The number of characters appearing in the `String` is both the size and capacity.

For `StringBuilder`, Java knows the size is likely to change as the object is used. When `StringBuilder` is constructed, it may start at the default capacity (which happens to be 16) or one of the programmer's choosing. In the example, we request a capacity of 5. At this point, the size is 0 since no characters have been added yet, but we have space for 5.

continues

(continued)

Next we add four characters. At this point, the size is 4 since four slots are taken. The capacity is still 5. Then we add three more characters. The size is now 7 since we have used up seven slots. Because the capacity wasn't large enough to store seven characters, Java automatically increased it for us.

```
StringBuilder sb = new StringBuilder(5);
```

0	1	2	3	4

```
sb.append("anim");
```

a	n	i	m	
0	1	2	3	4

```
sb.append("als");
```

a	n	i	m	a	l	s		
0	1	2	3	4	5	6	7	…

Important *StringBuilder* Methods

As with `String`, we aren't going to cover every single method in the `StringBuilder` class. These are the ones you might see on the exam.

charAt(), indexOf(), length(), and *substring()*

These four methods work exactly the same as in the `String` class. Be sure you can identify the output of this example:

```
StringBuilder sb = new StringBuilder("animals");
String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));
int len = sb.length();
char ch = sb.charAt(6);
System.out.println(sub + " " + len + " " + ch);
```

The correct answer is `anim 7 s`. The `indexOf()`method calls return 0 and 4, respectively. `substring()` returns the `String` starting with index 0 and ending right before index 4.

`length()` returns 7 because it is the number of characters in the `StringBuilder` rather than an index. Finally, `charAt()` returns the character at index 6. Here we do start with 0 because we are referring to indexes. If any of this doesn't sound familiar, go back and read the section on `String` again.

Notice that `substring()` returns a `String` rather than a `StringBuilder`. That is why *sb* is not changed. `substring()` is really just a method that inquires about where the substring happens to be.

append()

The `append()` method is by far the most frequently used method in `StringBuilder`. In fact, it is so frequently used that we just started using it without comment. Luckily, this method does just what it sounds like: it adds the parameter to the `StringBuilder` and returns a reference to the current `StringBuilder`. One of the method signatures is as follows:

```
StringBuilder append(String str)
```

Notice that we said *one* of the method signatures. There are more than 10 method signatures that look similar but that take different data types as parameters. All those methods are provided so you can write code like this:

```
StringBuilder sb = new StringBuilder().append(1).append('c');
sb.append("-").append(true);
System.out.println(sb);      // 1c-true
```

Nice method chaining, isn't it? `append()` is called directly after the constructor. By having all these method signatures, you can just call `append()` without having to convert your parameter to a `String` first.

insert()

The `insert()` method adds characters to the `StringBuilder` at the requested index and returns a reference to the current `StringBuilder`. Just like `append()`, there are lots of method signatures for different types. Here's one:

```
StringBuilder insert(int offset, String str)
```

Pay attention to the offset in these examples. It is the index where we want to insert the requested parameter.

```
3: StringBuilder sb = new StringBuilder("animals");
4: sb.insert(7, "-");                  // sb = animals-
5: sb.insert(0, "-");                  // sb = -animals-
6: sb.insert(4, "-");                  // sb = -ani-mals
7: System.out.println(sb);
```

Line 4 says to insert a dash at index 7, which happens to be the end of sequence of characters. Line 5 says to insert a dash at index 0, which happens to be the very beginning. Finally, line 6 says to insert a dash right before index 4. The exam creators will try to trip you up on this. As we add and remove characters, their indexes change. When you see a question dealing with such operations, draw what is going on so you won't be confused.

delete() and *deleteCharAt()*

The `delete()` method is the opposite of the `insert()` method. It removes characters from the sequence and returns a reference to the current `StringBuilder`. The `deleteCharAt()` method is convenient when you want to delete only one character. The method signatures are as follows:

```
StringBuilder delete(int start, int end)
StringBuilder deleteCharAt(int index)
```

The following code shows how to use these methods:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3);                  // sb = adef
sb.deleteCharAt(5);                  // throws an exception
```

First, we delete the characters starting with index 1 and ending right before index 3. This gives us `adef`. Next, we ask Java to delete the character at position 5. However, the remaining value is only four characters long, so it throws a `StringIndexOutOfBoundsException`.

reverse()

After all that, it's time for a nice, easy method. The `reverse()` method does just what it sounds like: it reverses the characters in the sequences and returns a reference to the current `StringBuilder`. The method signature is as follows:

```
StringBuilder reverse()
```

The following code shows how to use this method:

```
StringBuilder sb = new StringBuilder("ABC");
sb.reverse();
System.out.println(sb);
```

As expected, this prints `CBA`. This method isn't that interesting. Maybe the exam creators like to include it to encourage you to write down the value rather than relying on memory for indexes.

toString()

The last method converts a `StringBuilder` into a `String`. The method signature is as follows:

```
String toString()
```

The following code shows how to use this method:

```
String s = sb.toString();
```

Often `StringBuilder` is used internally for performance purposes but the end result needs to be a `String`. For example, maybe it needs to be passed to another method that is expecting a `String`.

StringBuilder vs. *StringBuffer*

When writing new code that concatenates a lot of `String` objects together, you should use `StringBuilder`. `StringBuilder` was added to Java in Java 5. If you come across older code, you will see `StringBuffer` used for this purpose. `StringBuffer` does the same thing but more slowly because it is thread safe. You'll learn about threads for the OCP exam. In theory, you don't need to know about `StringBuffer` on the exam at all. However, we bring this up anyway, since an older question might still be left on the exam.

Understanding Equality

In Chapter 2, you learned how to use == to compare numbers and that object references refer to the same object.

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two); // false
System.out.println(one == three); // true
```

Since this example isn't dealing with primitives, we know to look for whether the references are referring to the same object. *one* and *two* are both completely separate `StringBuilders`, giving us two objects. Therefore, the first print statement gives us `false`. *three* is more interesting. Remember how `StringBuilder` methods like to return the current reference for chaining? This means *one* and *three* both point to the same object and the second print statement gives us `true`.

Let's now visit the more complex and confusing scenario, `String` equality, made so in part because of the way the JVM reuses `String` literals:

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y);     // true
```

Remember that `Strings` are immutable and literals are pooled. The JVM created only one literal in memory. *x* and *y* both point to the same location in memory; therefore, the statement outputs true. It gets even trickier. Consider this code:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x == z); // false
```

In this example, we don't have two of the same `String` literal. Although *x* and *z* happen to evaluate to the same string, one is computed at runtime. Since it isn't the same at compile-time, a new `String` object is created.

You can even force the issue by creating a new `String`:

```
String x = new String("Hello World");
String y = "Hello World";
System.out.println(x == y); // false
```

Since you have specifically requested a different `String` object, the pooled value isn't shared.

The lesson is to never use `==` to compare `String` objects. The only time you should have to deal with `==` for `String`s is on the exam.

You saw earlier that you can say you want logical equality rather than object equality for `String` objects:

```
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x.equals(z)); // true
```

This works because the authors of the `String` class implemented a standard method called `equals` to check the values inside the `String` rather than the `String` itself. If a class doesn't have an `equals` method, Java determines whether the references point to the same object—which is exactly what `==` does. In case you are wondering, the authors of `StringBuilder` did not implement `equals()`. If you call `equals()` on two `StringBuilder` instances, it will check reference equality.

The exam will test you on your understanding of equality with objects they define too. For example:

```
1: public class Tiger {
2:   String name;
3:   public static void main(String[] args) {
4:     Tiger t1 = new Tiger();
5:     Tiger t2 = new Tiger();
6:     Tiger t3 = t1;
7:     System.out.println(t1 == t1); // true
8:     System.out.println(t1 == t2); // false
9:     System.out.println(t1.equals(t2)); // false
10:  } }
```

The first two statements check object reference equality. Line 7 prints `true` because we are comparing references to the same object. Line 8 prints `false` because the two object

references are different. Line 9 prints `false` since `Tiger` does not implement `equals()`. Don't worry—you aren't expected to know how to implement `equals()` for the OCA exam.

Understanding Java Arrays

Up to now, we've been referring to the `String` and `StringBuilder` classes as a "sequence of characters." This is true. They are implemented using an *array of characters*. An array is an area of memory on the heap with space for a designated number of elements. A `String` is implemented as an array with some methods that you might want to use when dealing with characters specifically. A `StringBuilder` is implemented as an array where the array object is replaced with a new bigger array object when it runs out of space to store all the characters. A big difference is that an array can be of any other Java type. If we didn't want to use a `String` for some reason, we could use an array of `char` primitives directly:

```
char[] letters;
```

This wouldn't be very convenient because we'd lose all the special properties `String` gives us, such as writing "Java". Keep in mind that *`letters`* is a reference variable and not a primitive. `char` is a primitive. But `char` is what goes into the array and not the type of the array itself. The array itself is of type `char[]`. You can mentally read the brackets (`[]`) as "array."

In other words, an array is an ordered list. It can contain duplicates. You will learn about data structures that cannot contain duplicates for the OCP exam. In this section, we'll look at creating an array of primitives and objects, sorting, searching, varargs, and multidimensional arrays.

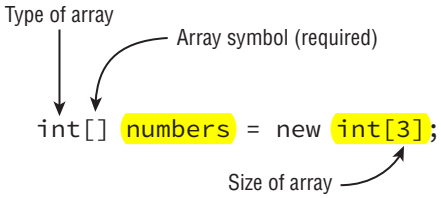
Creating an Array of Primitives

The most common way to create an array looks like this:

```
int[] numbers1 = new int[3];
```

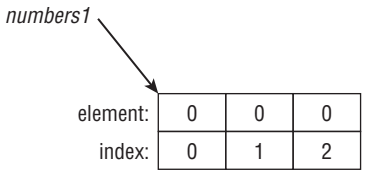
The basic parts are shown in Figure 3.2. It specifies the type of the array (`int`) and the size (3). The brackets tell you this is an array.

FIGURE 3.2 The basic structure of an array



When using this form to instantiate an array, set all the elements to the default value for that type. As you learned in Chapter 1, the default value of an `int` is 0. Since *`numbers1`* is a reference variable, it points to the array object, as shown in Figure 3.3. As you can see, the default value for all the elements is 0. Also, the indexes start with 0 and count up, just as they did for a `String`.

FIGURE 3.3 An empty array

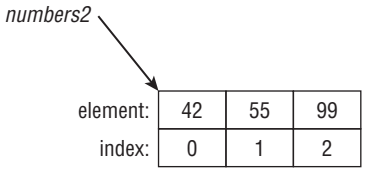


Another way to create an array is to specify all the elements it should start out with:

```
int[] numbers2 = new int[] {42, 55, 99};
```

In this example, we also create an `int` array of size 3. This time, we specify the initial values of those three elements instead of using the defaults. Figure 3.4 shows what this array looks like.

FIGURE 3.4 An initialized array



Java recognizes that this expression is redundant. Since you are specifying the type of the array on the left side of the equal sign, Java already knows the type. And since you are specifying the initial values, it already knows the size. As a shortcut, Java lets you write this:

```
int[] numbers2 = {42, 55, 99};
```

This approach is called an anonymous array. It is anonymous because you don't specify the type and size.

Finally, you can type the [] before or after the name, and adding a space is optional. This means that all four of these statements do the exact same thing:

```
int[] numAnimals;
int [] numAnimals2;
int numAnimals3[];
int numAnimals4 [];
```

Most people use the first one. You could see any of these on the exam, though, so get used to seeing the brackets in odd places.

Multiple "Arrays" in Declarations

What types of reference variables do you think the following code creates?

```
int[] ids, types;
```

The correct answer is two variables of type `int[]`. This seems logical enough. After all, `int a, b;` created two `int` variables. What about this example?

```
int ids[], types;
```

All we did was move the brackets, but it changed the behavior. This time we get one variable of type `int[]` and one variable of type `int`. Java sees this line of code and thinks something like this: "They want two variables of type `int`. The first one is called `ids[]`. This one is a `int[]` called *ids*. The second one is just called *types*. No brackets, so it is a regular integer."

Needless to say, you shouldn't write code that looks like this. But you do still need to understand it for the exam.

Creating an Array with Reference Variables

You can choose any Java type to be the type of the array. This includes classes you create yourself. Let's take a look at a built-in type with `String`:

```
public class ArrayType {
  public static void main(String args[]) {
    String [] bugs = { "cricket", "beetle", "ladybug" };
    String [] alias = bugs;
    System.out.println(bugs.equals(alias));     // true
    System.out.println(bugs.toString()); // [Ljava.lang.String;@160bc7c0
} }
```

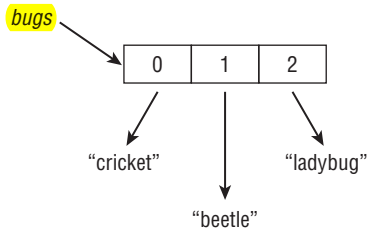
We can call `equals()` because an array is an object. It returns `true` because of reference equality. The `equals()` method on arrays does not look at the elements of the array. Remember, this would work even on an `int[]` too. `int` is a primitive; `int[]` is an object.

The second print statement is even more interesting. What on earth is `[Ljava.lang.String;@160bc7c0`? You don't have to know this for the exam, but `[L` means it is an array, `java.lang.String` is the reference type, and `160bc7c0` is the hash code.

Since Java 5, Java has provided a method that prints an array nicely: `java.util.Arrays.toString(bugs)` would print `[cricket, beetle, ladybug]`. The exam tends not to use it because most of the questions on arrays were written a long time ago. Regardless, this is a useful method when testing your own code.

Make sure you understand Figure 3.5. The array does not allocate space for the `String` objects. Instead, it allocates space for a reference to where the objects are really stored.

FIGURE 3.5 An array pointing to strings



As a quick review, what do you think this array points to?

```
class Names {
  String names[];
}
```

You got us. It was a review of Chapter 1 and not our discussion on arrays. The answer is `null`. The code never instantiated the array so it is just a reference variable to `null`. Let's try that again—what do you think this array points to?

```
class Names {
  String names[] = new String[2];
}
```

It is an array because it has brackets. It is an array of type `String` since that is the type mentioned in the declaration. It has two elements because the length is 2. Each of those two slots currently is `null`, but has the potential to point to a `String` object.

Remember casting from the previous chapter when you wanted to force a bigger type into a smaller type? You can do that with arrays too:

```
3: String[] strings = { "stringValue" };
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder();   // DOES NOT COMPILE
7: objects[0] = new StringBuilder();        // careful!
```

Line 3 creates an array of type `String`. Line 4 doesn't require a cast because `Object` is a broader type than `String`. On line 5, a cast is needed because we are moving to a more specific type. Line 6 doesn't compile because a `String[]` only allows `String` objects and `StringBuilder` is not a `String`.

Line 7 is where this gets interesting. From the point of view of the compiler, this is just fine. A `StringBuilder` object can clearly go in an `Object[]`. The problem is that we don't actually have an `Object[]`. We have a `String[]` referred to from an `Object[]` variable. At runtime, the code throws an `ArrayStoreException`. You don't need to memorize the name of this exception, but you do need to know that the code will throw an exception.

Using an Array

Now that we know how to create an array, let's try accessing one:

```
4: String[] mammals = {"monkey", "chimp", "donkey"};
5: System.out.println(mammals.length);          // 3
6: System.out.println(mammals[0]);              // monkey
7: System.out.println(mammals[1]);              // chimp
8: System.out.println(mammals[2]);              // donkey
```

Line 4 declares and initializes the array. Line 5 tells us how many elements the array can hold. The rest of the code prints the array. Notice elements are indexed starting with 0. This should be familiar from `String` and `StringBuilder`, which also start counting with 0. Those classes also counted `length` as the number of elements.

To make sure you understand how `length` works, what do you think this prints?

```
String[] birds = new String[6];
System.out.println(birds.length);
```

The answer is 6. Even though all 6 elements of the array are `null`, there are still 6 of them. `length` does not consider what is in the array; it only considers how many slots have been allocated.

It is very common to use a loop when reading from or writing to an array. This loop sets each element of `number` to 5 higher than the current index:

```
5: int[] numbers = new int[10];
6: for (int i = 0; i < numbers.length; i++)
7:  numbers[i] = i + 5;
```

Line 5 simply instantiates an array with 10 slots. Line 6 is a `for` loop using an extremely common pattern. It starts at index 0, which is where an array begins as well. It keeps going, one at a time, until it hits the end of the array. Line 7 sets the current element of *numbers*.

The exam will test whether you are being observant by trying to access elements that are not in the array. Can you tell why each of these throws an `ArrayIndexOutOfBoundsException` for our array of size 10?

```
numbers[10] = 3;
numbers[numbers.length] = 5;
for (int i = 0; i <= numbers.length; i++) numbers[i] = i + 5;
```

The first one is trying to see if you know that indexes start with 0. Since we have 10 elements in our array, this means only `numbers[0]` through `numbers[9]` are valid. The second example assumes you are clever enough to know 10 is invalid and disguises it by using the `length` variable. However, the length is always one more than the maximum valid index. Finally, the `for` loop incorrectly uses `<=` instead of `<`, which is also a way of referring to that 10th element.

Sorting

Java makes it easy to sort an array by providing a sort method—or rather, a bunch of sort methods. Just like `StringBuilder` allowed you to pass almost anything to `append()`, you can pass almost any array to `Arrays.sort()`.

`Arrays` is the first class provided by Java we have used that requires an import. To use it, you must have either of the following two statements in your class:

```
import java.util.*          // import whole package including Arrays
import java.util.Arrays;    // import just Arrays
```

There is one exception, although it doesn't come up often on the exam. You can write `java.util.Arrays` every time it is used in the class instead of specifying it as an import.

Remember that if you are shown a code snippet with a line number that doesn't begin with 1, you can assume the necessary imports are there. Similarly, you can assume the imports are present if you are shown a snippet of a method.

This simple example sorts three numbers:

```
int[] numbers = { 6, 9, 1 };
Arrays.sort(numbers);
for (int i = 0; i < numbers.length; i++)
L  System.out.print (numbers[i] +  " ");
```

The result is `1 6 9`, as you should expect it to be. Notice that we had to loop through the output to print the values in the array. Just printing the array variable directly would give the annoying hash of `[I@2bd9c3e7`.

Try this again with `String` types:

```
String[] strings = { "10", "9", "100" };
Arrays.sort(strings);
for (String string : strings)
  System.out.print(string + " ");
```

This time the result might not be what you expect. This code outputs `10 100 9`. The problem is that String sorts in alphabetic order, and `1` sorts before `9`. (Numbers sort before letters and uppercase sorts before lowercase, in case you were wondering.) For the OCP exam, you'll learn how to create custom sort orders using something called a comparator.

Did you notice we snuck in the enhanced `for` loop in this example? Since we aren't using the index, we don't need the traditional `for` loop. That won't stop the exam creators from using it, though, so we'll be sure to use both to keep you sharp!

Searching

Java also provides a convenient way to search—but only if the array is already sorted. Table 3.1 covers the rules for binary search.

TABLE 3.1 Binary search rules

Scenario	Result
Target element found in sorted array	Index of match
Target element not found in sorted array	Negative value showing one smaller than the negative of index, where a match needs to be inserted to preserve sorted order
Unsorted array	A surprise—this result isn't predictable

Let's try out these rules with an example:

```
3: int[] numbers = {2,4,6,8};
4: System.out.println(Arrays.binarySearch(numbers, 2)); // 0
5: System.out.println(Arrays.binarySearch(numbers, 4)); // 1
6: System.out.println(Arrays.binarySearch(numbers, 1)); // -1
7: System.out.println(Arrays.binarySearch(numbers, 3)); // -2
8: System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```

Take note of the fact that line 3 is a sorted array. If it weren't, we couldn't apply either of the other rules. Line 4 searches for the index of 2. The answer is index 0. Line 5 searches for the index of 4, which is 1.

Line 5 searches for the index of 1. Although 1 isn't in the list, the search can determine that it should be inserted at element 0 to preserve the sorted order. Since 0 already means something for array indexes, Java needs to subtract 1 to give us the answer of –1. Line 7 is similar. Although 3 isn't in the list, it would need to be inserted at element 1 to preserve the sorted order. We negate and subtract 1 for consistency, getting –1 –1, also known as –2. Finally, line 8 wants to tell us that 9 should be inserted at index 4. We again negate and subtract 1, getting –4 –1, also known as –5.

What do you think happens in this example?

```
5: int numbers = new int[] {3,2,1};
6: System.out.println(Arrays.binarySearch(numbers, 2));
7: System.out.println(Arrays.binarySearch(numbers, 3));
```

Note that on line 5, the array isn't sorted. This means the output will not be predictable. When testing this example, line 6 correctly gave 1 as the output. However, line 3 gave the wrong answer. The exam creators will not expect you to know what incorrect values come out. As soon as you see the array isn't sorted, look for an answer choice about unpredictable output.

On the exam, you need to know what a binary search returns in various scenarios. Oddly, you don't need to know why "binary" is in the name. In case you are curious, a binary search splits the array into two equal pieces (remember 2 is binary) and determines which half the target it is. It repeats this process until only one element is left.

Varargs

When creating an array yourself, it looks like what we've seen thus far. When one is passed to your method, there is another way it can look. Here are three examples with a `main()` method:

```
public static void main(String[] args)
public static void main(String args[])
public static void main(String... args) // varargs
```

The third example uses a syntax called varargs (variable arguments), which you saw in Chapter 1. You'll learn how to call a method using varargs in Chapter 4, "Methods and Encapsulation." For now, all you need to know is that you can use a variable defined using varargs as if it were a normal array. For example `args.length` and `args[0]` are legal.

Multidimensional Arrays

Arrays are objects, and of course array components can be objects. It doesn't take much time, rubbing those two facts together, to wonder if arrays can hold other arrays, and of course they can.

Creating a Multidimensional Array

Multiple array separators are all it takes to declare arrays with multiple dimensions. You can locate them with the type or variable name in the declaration, just as before:

```
int[][] vars1;          // 2D array
int vars2 [][];         // 2D array
int[] vars3[];          // 2D array
int[] vars4 [], space [][];  // a 2D AND a 3D array
```

The first two examples are nothing surprising and declare a two-dimensional (2D) array. The third example also declares a 2D array. There's no good reason to use this style other than to confuse readers of your code. The final example declares two arrays on the same line. Adding up the brackets, we see that the `vars4` is a 2D array and `space` is a 3D array. Again, there' no reason to use this style other than to confuse readers of your code. The exam creators like to try to confuse you, though. Luckily you are on to them and won't let this happen to you!

You can specify the size of your multidimensional array in the declaration if you like:

```
String [][] rectangle = new String[3][2];
```

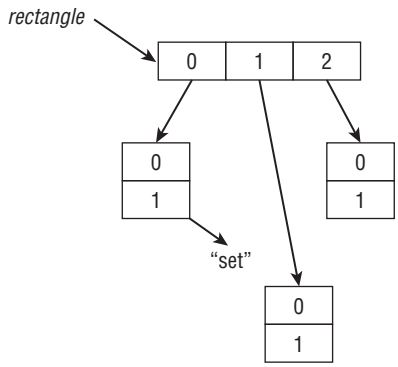
The result of this statement is an array rectangle with three elements, each of which refers to an array of two elements. You can think of the addressable range as [0][0] through [2][1], but don't think of it as a structure of addresses like [0,0] or [2,1].

Now suppose we set one of these values:

```
rectangle[0][1] = "set";
```

You can visualize the result as shown in Figure 3.6. This array is sparsely populated because it has a lot of null values. You can see that `rectangle` still points to an array of three elements and that we have three arrays of two elements. You can also follow the trail from reference to the one value pointing to a `String`. First you start at index 0 in the top array. Then you go to index 1 in the next array.

FIGURE 3.6 A sparsely populated multidimensional array

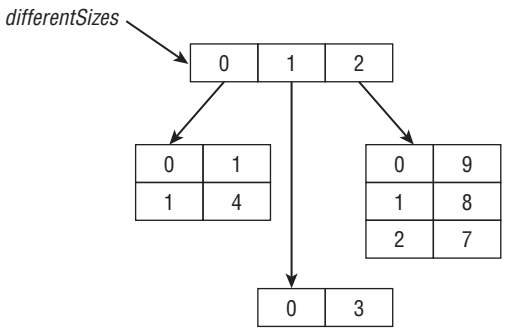


While that array happens to be rectangular in shape, an array doesn't need to be. Consider this one:

```
int[][] differentSize = {{1, 4}, {3}, {9,8,7}};
```

We still start with an array of three elements. However, this time the elements in the next level are all different sizes. One is of length 2, the next length 1, and the last length 3 (see Figure 3.7). This time the array is of primitives, so they are shown as if they are in the array themselves.

FIGURE 3.7 An asymmetric multidimensional array



Another way to create an asymmetric array is to initialize just an array's first dimension, and define the size of each array component in a separate statement:

```
int [][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```

This technique reveals what you really get with Java: arrays of arrays that, properly managed, offer a multidimensional effect.

Using a Multidimensional Array

The most common operation on a multidimensional array is to loop through it. This example prints out a 2D array:

```
int[][] twoD = new int[3][2];
for (int i = 0; i < twoD.length; i++) {
  for (int j = 0; j < twoD[i].length; j++)
    System.out.print(twoD[i][j] + " "); // print element
  System.out.println();                 // time for a new row
}
```

We have two loops here. The first uses index `i` and goes through the first subarray for `twoD`. The second uses a different loop variable `j`. It is very important these be different variable names so the loops don't get mixed up. The inner loop looks at how many elements are in the second-level array. The inner loop prints the element and leaves a space for readability. When the inner loop completes, the outer loop goes to a new line and repeats the process for the next element.

This entire exercise would be easier to read with the enhanced `for` loop.

```
for (int[] inner : twoD) {
  for (int num : inner)
```

```
    System.out.print(num + " ");
   System.out.println();
}
```

We'll grant you that it isn't fewer lines, but each line is less complex and there aren't any loop variables or terminating conditions to mix up.

Understanding an *ArrayList*

An array has one glaring shortcoming: you have to know how many elements will be in the array when you create it and then you are stuck with that choice. Just like a `StringBuilder`, `ArrayList` can change size at runtime as needed. Like an array, an `ArrayList` is an ordered sequence that allows duplicates.

As when we used `Arrays.sort`, `ArrayList` requires an import. To use it, you must have either of the following two statements in your class:

```
import java.util.*          // import whole package including ArrayList
import java.util.ArrayList; // import just ArrayList
```

Remember that if you are shown a code snippet with a line number that doesn't begin with 1, you can assume the necessary imports are there. Similarly, you can assume the imports are present if you are shown a snippet of a method.

In this section, we'll look at creating an `ArrayList`, common methods, autoboxing, conversion, and sorting.

Experienced programmers, take note: This section is simplified and doesn't cover a number of topics that are out of scope for the OCA exam.

Creating an *ArrayList*

As with `StringBuilder`, there are three ways to create an `ArrayList`:

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```

The first says to create an `ArrayList` containing space for the default number of elements but not to fill any slots yet. The second says to create an `ArrayList` containing a specific number of slots, but again not to assign any. The final example tells Java that we want to make a copy of another `ArrayList`. We copy both the size and contents of that `ArrayList`. Granted, *`list2`* is empty in this example so it isn't particularly interesting.

Although these are the only three constructors you need to know, you do need to learn some variants of it. The previous examples were the old pre–Java 5 way of creating an `ArrayList`. They still work and you still need to know they work. You also need to know

the new and improved way. Java 5 introduced *generics*, which allow you to specify the type of class that the `ArrayList` will contain.

```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
```

Java 5 allows you to tell the compiler what the type would be by specifying it between < and >. Starting in Java 7, you can even omit that type from the right side. The < and > are still required, though. This is called the diamond operator because <> looks like a diamond.

Just when you thought you knew everything about creating an `ArrayList`, there is one more thing you need to know. `ArrayList` implements an interface called `List`. In other words, an `ArrayList` is a `List`. You will learn about interfaces in Chapter 5. In the meantime, just know that you can store an `ArrayList` in a `List` reference variable but not vice versa. The reason is that `List` is an interface and interfaces can't be instantiated.

```
List<String> list6 = new ArrayList<>();
ArrayList<String> list7 = new List<>(); // DOES NOT COMPILE
```

Using an *ArrayList*

`ArrayList` has many methods, but you only need to know a handful of them—even fewer than you did for `String` and `StringBuilder`.

Before reading any further, you are going to see something new in the method signatures: a "class" named `E`. Don't worry—it isn't really a class. `E` is used by convention in generics to mean "any class that this array can hold." If you didn't specify a type when creating the `ArrayList`, `E` means `Object`. Otherwise, it means the class you put between `<` and `>`.

You should also know that `ArrayList` implements `toString()` so you can easily see the contents just by printing it. Arrays do not do produce such pretty output.

add()

The `add()` methods insert a new value in the `ArrayList`. The method signatures are as follows:

```
boolean add(E element)
void add(int index, E element)
```

Don't worry about the `boolean` return value. It always returns `true`. It is there because other classes in the collections family need a return value in the signature when adding an element.

Since `add()` is the most critical `ArrayList` method you need to know for the exam, we are going to show a few sets of examples for it. Let's start with the most straightforward case:

```
ArrayList list = new ArrayList();
list.add("hawk");          // [hawk]
```

```
list.add(Boolean.TRUE);    // [hawk, true]
System.out.println(list);  // [hawk, true]
```

`add()` does exactly what we expect: it stores the `String` in the no longer empty `ArrayList`. It then does the same thing for the `boolean`. This is okay because we didn't specify a type for `ArrayList`; therefore, the type is `Object`, which includes everything except primitives. It may not have been what we intended, but the compiler doesn't know that. Now, let's use generics to tell the compiler we only want to allow `String` objects in our `ArrayList`:

```
ArrayList<String> safer = new ArrayList<>();
safer.add("sparrow");
safer.add(Boolean.TRUE);    // DOES NOT COMPILE
```

This time the compiler knows that only `String` objects are allowed in and prevents the attempt to add a `boolean`. Now let's try adding multiple values to different positions.

```
4: List<String> birds = new ArrayList<>();
5: birds.add("hawk");            // [hawk]
6: birds.add(1, "robin");        // [hawk, robin]
7: birds.add(0, "blue jay");     // [blue jay, hawk, robin]
8: birds.add(1, "cardinal");     // [blue jay, cardinal, hawk, robin]
9: System.out.println(birds);    // [blue jay, cardinal, hawk, robin]
```

When a question has code that adds objects at indexed positions, draw it so that you won't lose track of which value is at which index. In this example, line 5 adds `"hawk"` to the end of `birds`. Then line 6 adds `"robin"` to index 1 of `birds`, which happens to be the end. Line 7 adds `"blue jay"` to index 0, which happens to be the beginning of `birds`. Finally, line 8 adds `"cardinal"` to index 1, which is now near the middle of `birds`.

remove()

The `remove()` methods remove the first matching value in the `ArrayList` or remove the element at a specified index. The method signatures are as follows:

```
boolean remove(Object object)
E remove(int index)
```

This time the `boolean` return value tells us whether a match was removed. The `E` return type is the element that actually got removed. The following shows how to use these methods:

```
3: List<String> birds = new ArrayList<>();
4: birds.add("hawk");     // [hawk]
5: birds.add("hawk");     // [hawk, hawk]
6: System.out.println(birds.remove("cardinal")); // prints false
7: System.out.println(birds.remove("hawk")); // prints true
```

```
8: System.out.println(birds.remove(0)); // prints hawk
9: System.out.println(birds);     // []
```

Line 6 tries to remove an element that is not in `birds`. It returns `false` because no such element is found. Line 7 tries to remove an element that is in `birds` and so returns `true`. Notice that it removes only one match. Line 8 removes the element at index 0, which is the last remaining element in the `ArrayList`.

Since calling `remove()` with an `int` uses the index, an index that doesn't exist will throw an exception. For example, `birds.remove(100)` throws an `IndexOutOfBoundsException`.

There is also a `removeIf()` method. We'll cover it in the next chapter because it uses lambda expressions (a topic in that chapter).

set()

The `set()` method changes one of the elements of the `ArrayList` without changing the size. The method signature is as follows:

```
E set(int index, E newElement)
```

The `E` return type is the element that got replaced. The following shows how to use this method:

```
15: List<String> birds = new ArrayList<>();
16: birds.add("hawk");                    // [hawk]
17: System.out.println(birds.size());     // 1
18: birds.set(0, "robin");                // [robin]
19: System.out.println(birds.size());     // 1
20: birds.set(1, "robin");                // IndexOutOfBoundsException
```

Line 16 adds one element to the array, making the size 1. Line 18 replaces that one element and the size stays at 1. Line 20 tries to replace an element that isn't in the `ArrayList`. Since the size is 1, the only valid index is 0. Java throws an exception because this isn't allowed.

isEmpty() and size()

The `isEmpty()` and `size()` methods look at how many of the slots are in use. The method signatures are as follows:

```
boolean isEmpty()
int size()
```

The following shows how to use these methods:

```
System.out.println(birds.isEmpty());     // true
System.out.println(birds.size());     // 0
birds.add("hawk");                    // [hawk]
birds.add("hawk");                    // [hawk, hawk]
```

```
System.out.println(birds.isEmpty());     // false
System.out.println(birds.size());     // 2
```

At the beginning, `birds` has a size of 0 and is empty. It has a capacity that is greater than 0. However, as with `StringBuilder`, we don't use the capacity in determining size or length. After adding elements, the size becomes positive and it is no longer empty.

clear()

The `clear()` method provides an easy way to discard all elements of the `ArrayList`. The method signature is as follows:

```
void clear()
```

The following shows how to use this method:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");                    // [hawk]
birds.add("hawk");                    // [hawk, hawk]
System.out.println(birds.isEmpty());     // false
System.out.println(birds.size());     // 2
birds.clear();                         // []
System.out.println(birds.isEmpty());     // true
System.out.println(birds.size());     // 0
```

After we call `clear()`, `birds` is back to being an empty `ArrayList` of size 0.

contains()

The `contains()` method checks whether a certain value is in the `ArrayList`. The method signature is as follows:

```
boolean contains(Object object)
```

The following shows how to use this method:

```
List<String> birds = new ArrayList<>();
birds.add("hawk");                          // [hawk]
System.out.println(birds.contains("hawk")); // true
System.out.println(birds.contains("robin")); // false
```

This method calls `equals()` on each element of the `ArrayList` to see whether there are any matches. Since `String` implements `equals()`, this works out well.

equals()

Finally, `ArrayList` has a custom implementation of `equals()` so you can compare two lists to see if they contain the same elements in the same order.

```
boolean equals(Object object)
```

The following shows an example:

```
31: List<String> one = new ArrayList<>();
32: List<String> two = new ArrayList<>();
33: System.out.println(one.equals(two));      // true
34: one.add("a");                            // [a]
35: System.out.println(one.equals(two));     // false
36: two.add("a");                            // [a]
37: System.out.println(one.equals(two));     // true
38: one.add("b");                            // [a,b]
39: two.add(0, "b");                        // [b,a]
40: System.out.println(one.equals(two));     // false
```

On line 33, the two `ArrayList` objects are equal. An empty list is certainly the same elements in the same order. On line 35, the `ArrayList` objects are not equal because the size is different. On line 37, they are equal again because the same one element is in each. On line 40, they are not equal. The size is the same and the values are the same, but they are not in the same order.

Wrapper Classes

Up to now, we've only put `String` objects in the `ArrayList`. What happens if we want to put primitives in? Each primitive type has a wrapper class, which is an object type that corresponds to the primitive. Table 3.2 lists all the wrapper classes along with the constructor for each.

TABLE 3.2 Wrapper classes

Primitive type	Wrapper class	Example of constructing
boolean	Boolean	`new Boolean(true)`
byte	Byte	`new Byte((byte) 1)`
short	Short	`new Short((short) 1)`
int	Integer	`new Integer(1)`
long	Long	`new Long(1)`
float	Float	`new Float(1.0)`
double	Double	`new Double(1.0)`
char	Character	`new Character('c')`

The wrapper classes also have a method that converts back to a primitive. You don't need to know much about the constructors or `intValue()` type methods for the exam because autoboxing has removed the need for them (see the next section). You might encounter this syntax on questions that have been on the exam for many years. However, you just need to be able to read the code and not look for tricks in it.

There are also methods for converting a `String` to a primitive or wrapper class. You do need to know these methods. The parse methods, such as `parseInt()`, return a primitive, and the `valueOf()` method returns a wrapper class. This is easy to remember because the name of the returned primitive is in the method name. For example:

```
int primitive = Integer.parseInt("123");
Integer wrapper = Integer.valueOf("123");
```

The first line converts a `String` to an `int` primitive. The second converts a `String` to an `Integer` wrapper class. If the `String` passed in is not valid for the given type, Java throws an exception. In these examples, letters and dots are not valid for an integer value:

```
int bad1 = Integer.parseInt("a");          // throws NumberFormatException
Integer bad2 = Integer.valueOf("123.45");  // throws NumberFormatException
```

Before you worry, the exam won't make you recognize that the method `parseInt()` is used rather than `parseInteger()`. You simply need to be able to recognize the methods when put in front of you. Also, the `Character` class doesn't participate in the `parse/valueOf` methods. Since a `String` is made up of characters, you can just call `charAt()` normally.

Table 3.3 lists the methods you need to recognize for creating a primitive or wrapper class object from a `String`. In real coding, you won't be so concerned which is returned from each method due to autoboxing.

TABLE 3.3 Converting from a `String`

Wrapper class	Converting `String` to primitive	Converting `String` to wrapper class
Boolean	`Boolean.parseBoolean("true");`	`Boolean.valueOf("TRUE");`
Byte	`Byte.parseByte("1");`	`Byte.valueOf("2");`
Short	`Short.parseShort("1");`	`Short.valueOf("2");`
Integer	`Integer.parseInt("1");`	`Integer.valueOf("2");`
Long	`Long.parseLong("1");`	`Long.valueOf("2");`
Float	`Float.parseFloat("1");`	`Float.valueOf("2.2");`
Double	`Double.parseDouble("1");`	`Double.valueOf("2.2");`
Character	None	None

Autoboxing

Why won't you need to be concerned with whether a primitive or wrapper class is returned, you ask? Since Java 5, you can just type the primitive value and Java will convert it to the relevant wrapper class for you. This is called *autoboxing*. Let's look at an example:

```
4: List<Double> weights = new ArrayList<>();
5: weights.add(50.5);                // [50.5]
6: weights.add(new Double(60));      // [50.5, 60.0]
7: weights.remove(50.5);                // [60.0]
8: double first = weights.get(0);     // 60.0
```

Line 5 autoboxes the `double` primitive into a `Double` object and adds that to the `List`. Line 6 shows that you can still write code the long way and pass in a wrapper object. Line 7 again autoboxes into the wrapper object and passes it to `remove()`. Line 8 retrieves the `Double` and unboxes it into a `double` primitive.

What do you think happens if you try to unbox a `null`?

```
3: List<Integer> heights = new ArrayList<>();
4: heights.add(null);
5: int h = heights.get(0);           // NullPointerException
```

On line 4, we add a `null` to the list. This is legal because a `null` reference can be assigned to any reference variable. On line 5, we try to unbox that `null` to an `int` primitive. This is a problem. Java tries to get the `int` value of `null`. Since calling any method on `null` gives a `NullPointerException`, that is just what we get. Be careful when you see `null` in relation to autoboxing.

Be careful when autoboxing into `Integer`. What do you think this code outputs?

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.remove(1);
System.out.println(numbers);
```

It actually outputs 1. After adding the two values, the `List` contains [1, 2]. We then request the element with index 1 be removed. That's right: index 1. Because there's already a `remove()` method that takes an `int` parameter, Java calls that method rather than autoboxing. If you want to remove the 2, you can write `numbers.remove(new Integer(2))` to force wrapper class use.

Converting Between *array* and *List*

You should know how to convert between an array and an `ArrayList`. Let's start with turning an `ArrayList` into an array:

```
3: List<String> list = new ArrayList<>();
4: list.add("hawk");
```

```
5: list.add("robin");
6: Object[] objectArray = list.toArray();
7: System.out.println(objectArray.length);     // 2
8: String[] stringArray = list.toArray(new String[0]);
9: System.out.println(stringArray.length);     // 2
```

Line 6 shows that an `ArrayList` knows how to convert itself to an array. The only problem is that it defaults to an array of class `Object`. This isn't usually what you want. Line 8 specifies the type of the array and does what we actually want. The advantage of specifying a size of 0 for the parameter is that Java will create a new array of the proper size for the return value. If you like, you can suggest a larger array to be used instead. If the `ArrayList` fits in that array, it will be returned. Otherwise, a new one will be created.

Converting from an array to a `List` is more interesting. The original array and created array backed `List` are linked. When a change is made to one, it is available in the other. It is a fixed-size list and is also known a *backed* `List` because the array changes with it. Pay careful attention to the values here:

```
20: String[] array = { "hawk", "robin" };     // [hawk, robin]
21: List<String> list = Arrays.asList(array); // returns fixed size list
22: System.out.println(list.size());     // 2
23: list.set(1, "test");          // [hawk, test]
24: array[0] = "new";               // [new, test]
25: for (String b : array) System.out.print(b + " "); // new test
26: list.remove(1);     // throws UnsupportedOperation Exception
```

Line 21 converts the array to a `List`. Note that it isn't the `java.util.ArrayList` we've grown used to. It is a fixed-size, backed version of a `List`. Line 23 is okay because `set()` merely replaces an existing value. It updates both *`array`* and *`list`* because they point to the same data store. Line 24 also changes both *`array`* and *`list`*. Line 25 shows the array has changed to `new test`. Line 26 throws an exception because we are not allowed to change the size of the list.

A Cool Trick with Varargs

This topic isn't on the exam, but merging varargs with `ArrayList` conversion allows you to create an `ArrayList` in a cool way:

```
List<String> list = Arrays.asList("one", "two");
```

`asList()` takes varargs, which let you pass in an array or just type out the `String` values. This is handy when testing because you can easily create and populate a `List` on one line.

Sorting

Sorting an `ArrayList` is very similar to sorting an array. You just use a different helper class:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(99);
numbers.add(5);
numbers.add(81);
Collections.sort(numbers);
System.out.println(numbers); [5, 81, 99]
```

As you can see, the numbers got sorted, just like you'd expect. Isn't it nice to have something that works just like you think it will?

Working with Dates and Times

In Java 8, Oracle completely revamped how we work with dates and times. You can still write code the "old way," but those classes aren't on the exam. We'll mention the "old way" in real-world scenarios so that you can learn the "new way" more easily if you first learned Java before version 8. Even if you are learning Java starting with version 8, this will help you when you need to read older code. Just know that the "old way" is not on the exam.

As with an `ArrayList`, you need an import statement to work with the date and time classes. Most of them are in the `java.time` package. To use it, add this `import` to your program:

```
import java.time.*;          // import time classes
```

In the following sections, we'll look at creating, manipulating, and formatting dates and times.

Creating Dates and Times

In the real world, we usually talk about dates and time zones as if the other person is located near us. For example, if you say "I'll call you at 11:00 on Tuesday morning," we assume that 11:00 means the same thing to both of us. But if you live in New York and we live in California, you need to be more specific. California is three hours earlier than New York because the states are in different time zones. You would instead say, "I'll call you at 11:00 EST on Tuesday morning." Luckily, the exam doesn't cover time zones, so discussing dates and times is easier.

When working with dates and times, the first thing to do is decide how much information you need. The exam gives you three choices:

`LocalDate` Contains just a date—no time and no time zone. A good example of `LocalDate` is your birthday this year. It is your birthday for a full day regardless of what time it is.

`LocalTime` Contains just a time—no date and no time zone. A good example of `LocalTime` is midnight. It is midnight at the same time every day.

`LocalDateTime` Contains both a date and time but no time zone. A good example of `LocalDateTime` is "the stroke of midnight on New Year's." Midnight on January 2 isn't nearly as special, and clearly an hour after midnight isn't as special either.

Oracle recommends avoiding time zones unless you really need them. Try to act as if everyone is in the same time zone when you can. If you do need to communicate across time zones, `ZonedDateTime` handles them.

Ready to create your first date and time objects?

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
```

Each of the three classes has a static method called `now()` that gives the current date and time. Your output is going to depend on what date/time you run it and where you live. The authors live in the United States, making the output look like the following when run on January 20 at 12:45 p.m.:

```
2015-01-20
12:45:18.401
2015-01-20T12:45:18.401
```

The key is to notice the type of information in the output. The first one contains only a date and no time. The second contains only a time and no date. This time displays hours, minutes, seconds, and nanoseconds. The third contains both date and time. Java uses `T` to separate the date and time when converting `LocalDateTime` to a `String`.

Wait—I Don't Live in the United States

The exam recognizes that exam takers live all over the world and will not ask you about the details of United States date and time formats.

In the United States, the month is written before the date. The exam won't ask you about the difference between 02/03/2015 and 03/02/2015. That would be mean and not internationally friendly, and it would be testing your knowledge of United States dates rather than your knowledge of Java. That said, our examples do use United States date and time formats as will the questions on the exam. Just remember that the month comes before the date. Also, Java tends to use a 24-hour clock even though the United States uses a 12-hour clock with a.m./p.m.

Now that you know how to create the current date and time, let's look at other specific dates and times. To begin, let's create just a date with no time. Both of these examples create the same date:

```
LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);
LocalDate date2 = LocalDate.of(2015, 1, 20);
```

Both pass in the year, month, and date. Although it is good to use the `Month` constants (to make the code easier to read), you can pass the `int` number of the month directly. Just use the number of the month the same way you would if you were writing the date in real life.

The method signatures are as follows:

```
public static LocalDate of(int year, int month, int dayOfMonth)
public static LocalDate of(int year, Month month, int dayOfMonth)
```

Month is a special type of class called an enum. You don't need to know about enums on the OCA exam and can just treat them as constants.

Up to now, we've been like a broken record telling you that Java counts starting with 0. Well, months are an exception. For months in the new date and time methods, Java counts starting from 1 like we human beings do.

When creating a time, you can choose how detailed you want to be. You can specify just the hour and minute, or you can add the number of seconds. You can even add nanoseconds if you want to be very precise. (A nanosecond is a billionth of a second—you probably won't need to be that specific.)

```
LocalTime time1 =  LocalTime.of(6, 15);              // hour and minute
LocalTime time2 =  LocalTime.of(6, 15, 30);          // + seconds
LocalTime time3 =  LocalTime.of(6, 15, 30, 200);     // + nanoseconds
```

These three times are all different but within a minute of each other. The method signatures are as follows:

```
public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second, int nanos)
```

Finally, we can combine dates and times:

```
LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 20, 6, 15, 30);
LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);
```

The first line of code shows how you can specify all the information about the `LocalDateTime` right in the same line. There are many method signatures allowing you

to specify different things. Having that many numbers in a row gets to be hard to read, though. The second line of code shows how you can create `LocalDate` and `LocalTime` objects separately first and then combine them to create a `LocalDateTime` object.

This time there are a lot of method signatures since there are more combinations. The method signatures are as follows:

```
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, int month,
  int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, Month month,
  int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime)
```

Did you notice that we did not use a constructor in any of the examples? The date and time classes have private constructors to force you to use the static methods. The exam creators may try to throw something like this at you:

```
LocalDate d = new LocalDate(); // DOES NOT COMPILE
```

Don't fall for this. You are not allowed to construct a date or time object directly. Another trick is to see what happens when you pass invalid numbers to `of()`. For example:

```
LocalDate.of(2015, Month.JANUARY, 32)     // throws DateTimeException
```

You don't need to know the exact exception that's thrown, but it's a clear one:

```
java.time.DateTimeException: Invalid value for DayOfMonth
  (valid values 1 - 28/31): 32
```

Real World Scenario

Creating Dates in Java 7 and Earlier

You can see some of the problems with the "old way" in the following table. There wasn't a way to specify just a date without the time. The `Date` class represented both the date and time whether you wanted it to or not. Trying to create a specific date required more code than it should have. Month indexes were 0 based instead of 1 based, which was confusing.

continues

continued

There's an old way to create a date. In Java 1.1, you created a specific `Date` with this: `Date jan = new Date(2015, Calendar.JANUARY, 1);`. You could use the `Calendar` class beginning with Java 1.2. `Date` exists mainly for backward compatibility and so that `Calendar` can work with code—making the "new way" the third way. The "new way" is much better so it looks like this is a case of the third time is the charm!

	Old way	New way (Java 8 and later)
Importing	`import java.util.*;`	`import java.time.*;`
Creating an object with the current date	`Date d = new Date();`	`LocalDate d = LocalDate.now();`
Creating an object with the current date and time	`Date d = new Date();`	`LocalDateTime dt = LocalDateTime.now();`

Creating an object representing January 1, 2015	`Calendar c = Calendar.getInstance();`
	`c.set(2015, Calendar.JANUARY, 1);`
	`Date jan = c.getTime();`
	or
	`Calendar c = new GregorianCalendar(2015, Calendar.JANUARY, 1);`
	`Date jan = c.getTime();`
		`LocalDate jan = LocalDate.of(2015, Month.JANUARY, 1);`

Creating January 1, 2015 without the constant	`Calendar c = Calendar.getInstance();`
	`c.set(2015, 0, 1);`
	`Date jan = c.getTime();`
		`LocalDate jan = LocalDate.of(2015, 1, 1)`

Manipulating Dates and Times

Adding to a date is easy. The date and time classes are immutable, just like `String` was. This means that we need to remember to assign the results of these methods to a reference variable so they are not lost.

```
12: LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
13: System.out.println(date);          // 2014-01-20
14: date = date.plusDays(2);
15: System.out.println(date);          // 2014-01-22
16: date = date.plusWeeks(1);
```

```
17: System.out.println(date);          // 2014-01-29
18: date = date.plusMonths(1);
19: System.out.println(date);          // 2014-02-28
20: date = date.plusYears(5);
21: System.out.println(date);          // 2019-02-28
```

This code is nice because it does just what it sounds like. We start out with January 20, 2014. On line 14, we add two days to it and reassign it to our reference variable. On line 16, we add a week. This method allows us to write clearer code than `plusDays(7)`. Now `date` is January 29, 2014. On line 18, we add a month. This would bring us to February 29, 2014. However, 2014 is not a leap year. (2012 and 2016 are leap years.) Java is smart enough to realize February 29, 2014 does not exist and gives us February 28, 2014 instead. Finally, line 20 adds five years.

There are also nice easy methods to go backward in time. This time, let's work with `LocalDateTime`.

```
22: LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
23: LocalTime time = LocalTime.of(5, 15);
24: LocalDateTime dateTime = LocalDateTime.of(date, time);
25: System.out.println(dateTime);          // 2020-01-20T05:15
26: dateTime = dateTime.minusDays(1);
27: System.out.println(dateTime);          // 2020-01-19T05:15
28: dateTime = dateTime.minusHours(10);
29: System.out.println(dateTime);          // 2020-01-18T19:15
30: dateTime = dateTime.minusSeconds(30);
31: System.out.println(dateTime);          // 2020-01-18T19:14:30
```

Line 25 prints the original date of January 20, 2020 at 5:15 a.m. Line 26 subtracts a full day, bringing us to January 19, 2020 at 5:15 a.m. Line 28 subtracts 10 hours, showing that the date will change if the hours cause it to and brings us to January 18, 2020 at 19:15 (7:15 p.m.). Finally, line 30 subtracts 30 seconds. We see that all of a sudden the display value starts displaying seconds. Java is smart enough to hide the seconds and nanoseconds when we aren't using them.

It is common for date and time methods to be chained. For example, without the print statements, the previous example could be rewritten as follows:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date2, time)
   .minusDays(1).minusHours(10).minusSeconds(30);
```

When you have a lot of manipulations to make, this chaining comes in handy. There are two ways the exam creators can try to trick you. What do you think this prints?

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date.plusDays(10);
System.out.println(date);
```

It prints January 20, 2020. Adding 10 days was useless because we ignored the result. Whenever you see immutable types, pay attention to make sure the return value of a method call isn't ignored.

The exam also may test to see if you remember what each of the date and time objects includes. Do you see what is wrong here?

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date = date.plusMinutes(1);     // DOES NOT COMPILE
```

`LocalDate` does not contain time. This means you cannot add minutes to it. This can be tricky in a chained sequence of additions/subtraction operations, so make sure you know which methods in Table 3.4 can be called on which of the three objects.

TABLE 3.4 Methods in `LocalDate`, `LocalTime`, and `LocalDateTime`

	Can call on `LocalDate`?	Can call on `LocalTime`?	Can call on `LocalDateTime`?
`plusYears`/`minusYears`	Yes	No	Yes
`plusMonths`/`minusMonths`	Yes	No	Yes
`plusWeeks`/`minusWeeks`	Yes	No	Yes
`plusDays`/`minusDays`	Yes	No	Yes
`plusHours`/`minusHours`	No	Yes	Yes
`plusMinutes`/`minusMinutes`	No	Yes	Yes
`plusSeconds`/`minusSeconds`	No	Yes	Yes
`plusNanos`/`minusNanos`	No	Yes	Yes

Manipulating Dates in Java 7 and Earlier

As you look at all the code in the following table to do time calculations in the "old way," you can see why Java needed to revamp the date and time APIs! The "old way" took a lot of code to do something simple.

	Old way	New way (Java 8 and later)

Adding a day	`public Date addDay(Date date) {`
	`  Calendar cal = Calendar`
	`  .getInstance();`
	`  cal.setTime(date);`
	`  cal.add(Calendar.DATE, 1);`
	`  return cal.getTime();`
	`}`
		`public LocalDate`
		`addDay(LocalDate date) {`
		`  return date.`
		`  plusDays(1);`
		`}`

Subtracting a day	`public Date subtractDay(Date date)`
	`{`
	`  Calendar cal = Calendar`
	`.getInstance();`
	`  cal.setTime(date);`
	`  cal.add(Calendar.DATE, -1);`
	`  return cal.getTime();`
	`}`
		`public LocalDate`
		`subtractDay(LocalDate`
		`date) {`
		`  return date.`
		`  minusDays(1);`
		`}`

Working with Periods

Now we know enough to do something fun with dates! Our zoo performs animal enrichment activities to give the animals something fun to do. The head zookeeper has decided to switch the toys every month. This system will continue for three months to see how it works out.

```
public static void main(String[] args) {
  LocalDate start = LocalDate.of(2015, Month.JANUARY, 1);
  LocalDate end = LocalDate.of(2015, Month.MARCH, 30);
  performAnimalEnrichment(start, end);
}
private static void performAnimalEnrichment(LocalDate start, LocalDate end) {
  LocalDate upTo = start;
  while (upTo.isBefore(end)) {          // check if still before end
   System.out.println("give new toy: " + upTo);
   upTo  = upTo.plusMonths(1);                    // add a month
  }}
```

This code works fine. It adds a month to the date until it hits the end date. The problem is that this method can't be reused. Our zookeeper wants to try different schedules to see which works best.

Converting to a `long`

`LocalDate` and `LocalDateTime` have a method to convert them into `long` equivalents in relation to 1970. What's special about 1970? That's what UNIX started using for date standards, so Java reused it. And don't worry—you don't have to memorize the names for the exam.

- `LocalDate` has `toEpochDay()`, which is the number of days since January 1, 1970.
- `LocalDateTime` has `toEpochTime()`, which is the number of seconds since January 1, 1970.
- `LocalTime` does not have an epoch method. Since it represents a time that occurs on any date, it doesn't make sense to compare it in 1970. Although the exam pretends time zones don't exist, you may be wondering if this special January 1, 1970 is in a specific time zone. The answer is yes. This special time refers to when it was January 1, 1970 in GMT (Greenwich Mean Time). Greenwich is in England and GMT does not participate in daylight savings time. This makes it a good reference point. (Again, you don't have to know about GMT for the exam.)

Luckily, Java has a `Period` class that we can pass in. This code does the same thing as the previous example:

```
public static void main(String[] args) {
  LocalDate start = LocalDate.of(2015, Month.JANUARY, 1);
  LocalDate end = LocalDate.of(2015, Month.MARCH, 30);
  Period period = Period.ofMonths(1);               // create a period
  performAnimalEnrichment(start, end, period);
}
private static void performAnimalEnrichment(LocalDate start, LocalDate end,
  Period period) {               // uses the generic period
  LocalDate upTo = start;
  while (upTo.isBefore(end)) {
    System.out.println("give new toy: " + upTo);
    upTo = upTo.plus(period);     // adds the period
  }}
```

The method can add an arbitrary period of time that gets passed in. This allows us to reuse the same method for different periods of time as our zookeeper changes her mind.

There are five ways to create a `Period` class:

```
Period annually = Period.ofYears(1);              // every 1 year
Period quarterly = Period.ofMonths(3);              // every 3 months
```

```
Period everyThreeWeeks = Period.ofWeeks(3);          // every 3 weeks
Period everyOtherDay = Period.ofDays(2);           // every 2 days
Period everyYearAndAWeek = Period.of(1, 0, 7);          // every year and 7 days
```

There's one catch. You cannot chain methods when creating a `Period`. The following code looks like it is equivalent to the `everyYearAndAWeek` example, but it's not. Only the last method is used because the `Period.ofXXX` methods are static methods.

```
Period wrong = Period.ofYears(1).ofWeeks(1);          // every week
```

This tricky code is really like writing the following:

```
Period wrong = Period.ofYears(1);
wrong = Period.ofWeeks(7);
```

This is clearly not what you intended! That's why the `of()` method allows us to pass in the number of years, months, and days. They are all included in the same period. You will get a compiler warning about this. Compiler warnings tell you something is wrong or suspicious without failing compilation.

You've probably noticed by now that a `Period` is a day or more of time. There is also `Duration`, which is intended for smaller units of time. For `Duration`, you can specify the number of days, hours, minutes, seconds, or nanoseconds. And yes, you could pass 365 days to make a year, but you really shouldn't—that's what `Period` is for. `Duration` isn't on the exam since it roughly works the same way as `Period`. It's good to know it exists, though.

The last thing to know about `Period` is what objects it can be used with. Let's look at some code:

```
3: LocalDate date = LocalDate.of(2015, 1, 20);
4: LocalTime time = LocalTime.of(6, 15);
5: LocalDateTime dateTime = LocalDateTime.of(date, time);
6: Period period = Period.ofMonths(1);
7: System.out.println(date.plus(period));          // 2015-02-20
8: System.out.println(dateTime.plus(period));          // 2015-02-20T06:15
9: System.out.println(time.plus(period));   // UnsupportedTemporalTypeException
```

Lines 7 and 8 work as expected. They add a month to January 20, 2015, giving us February 20, 2015. The first has only the date, and the second has both the date and time.

Line 9 attempts to add a month to an object that only has a time. This won't work. Java throws an exception and complains that we attempt to use an `Unsupported unit: Months.`

As you can see, you'll have to pay attention to the type of date and time objects every place you see them.

Formatting Dates and Times

The date and time classes support many methods to get data out of them:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
System.out.println(date.getDayOfWeek());     // MONDAY
System.out.println(date.getMonth());          // JANUARY
System.out.println(date.getYear());          // 2020
System.out.println(date.getDayOfYear());     // 20
```

We could use this information to display information about the date. However, it would be more work than necessary. Java provides a class called `DateTimeFormatter` to help us out. Unlike the `LocalDateTime` class, `DateTimeFormatter` can be used to format any type of date and/or time object. What changes is the format. `DateTimeFormatter` is in the package `java.time.format`.

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);System.out.println(date
.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

ISO is a standard for dates. The output of the previous code looks like this:

```
2020-01-20
11:12:34
2020-01-20T11:12:34
```

This is a reasonable way for computers to communicate, but probably not how you want to output the date and time in your program. Luckily there are some predefined formats that are more useful:

```
DateTimeFormatter shortDateTime =
  DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(shortDateTime.format(dateTime));     // 1/20/20
System.out.println(shortDateTime.format(date));      // 1/20/20
System.out.println(
  shortDateTime.format(time)); // UnsupportedTemporalTypeException
```

Here we say we want a localized formatter in the predefined short format. The last line throws an exception because a time cannot be formatted as a date. The `format()` method is declared on both the formatter objects and the date/time objects, allowing you to reference the objects in either order. The following statements print exactly the same thing as the previous code:

```
DateTimeFormatter shortDateTime =
  DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(dateTime.format(shortDateTime));
System.out.println(date.format(shortDateTime));
System.out.println(time.format(shortDateTime));
```

In this book, we'll change around the orders to get you used to seeing it both ways. Table 3.5 shows the legal and illegal localized formatting methods.

TABLE 3.5 ofLocalized methods

`DateTimeFormatter f = DateTimeFormatter.____ (FormatStyle.SHORT);`	**Calling `f.format (localDate)`**	**Calling `f.format (localDateTime)`**	**Calling `f.format (localTime)`**
`ofLocalizedDate`	Legal – shows whole object	Legal – shows just date part	Throws runtime exception
`ofLocalizedDateTime`	Throws runtime exception	Legal – shows whole object	Throws runtime exception
`ofLocalizedTime`	Throws runtime exception	Legal – shows just time part	Legal – shows whole object

There are two predefined formats that can show up on the exam: `SHORT` and `MEDIUM`. The other predefined formats involve time zones, which are not on the exam.

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);

DateTimeFormatter shortF = DateTimeFormatter
   .ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter mediumF = DateTimeFormatter
  .ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(shortF.format(dateTime));     // 1/20/20 11:12 AM
System.out.println(mediumF.format(dateTime));     // Jan 20, 2020 11:12:34 AM
```

If you don't want to use one of the predefined formats, you can create your own. For example, this code spells out the month:

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
System.out.println(dateTime.format(f));     // January 20, 2020, 11:12
```

Before we look at the syntax, know you are not expected to memorize what different numbers of each symbol mean. The most you will need to do is recognize the date and time pieces.

MMMM `M` represents the month. The more `M`s you have, the more verbose the Java output. For example, `M` outputs 1, `MM` outputs 01, `MMM` outputs Jan, and `MMMM` outputs January.

dd `d` represents the date in the month. As with month, the more `d`s you have, the more verbose the Java output. `dd` means to include the leading zero for a single-digit month.

, Use **,** if you want to output a comma (this also appears after the year).

yyyy `y` represents the year. `yy` outputs a two-digit year and `yyyy` outputs a four-digit year.

hh `h` represents the hour. Use `hh` to include the leading zero if you're outputting a single-digit hour.

: Use **:** if you want to output a colon.

mm `m` represents the minute.

Real World Scenario

Formatting Dates in Java 7 and Earlier

Formatting is roughly equivalent to the "old way"; it just uses a different class.

	Old way	New way (Java 8 and later)
Formatting the times	`SimpleDateFormat sf = new SimpleDateFormat("hh:mm");` `sf.format(jan3);`	`DateTimeFormatter f = DateTimeFormatter.ofPattern("hh:mm");` `dt.format(f);`

Let's do a quick review. Can you figure out which of these lines will throw an exception?

```
4: DateTimeFormatter f = DateTimeFormatter.ofPattern("hh:mm");
5: f.format(dateTime);
6: f.format(date);
7: f.format(time);
```

If you get this question on the exam, think about what the symbols represent. We have `h` for hour and `m` for minute. Remember `M` (uppercase) is month and `m` (lowercase) is minute. We can only use this formatter with objects containing times. Therefore, line 6 will throw an exception.

Parsing Dates and Times

Now that you know how to convert a date or time to a formatted `String`, you'll find it easy to convert a `String` to a date or time. Just like the `format()` method, the `parse()` method takes a formatter as well. If you don't specify one, it uses the default for that type.

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);
LocalTime time = LocalTime.parse("11:22");
System.out.println(date);          // 2015-01-02
System.out.println(time);          // 11:22
```

Here we show using both a custom formatter and a default value. This isn't common, but you might have to read code that looks like this on the exam. Parsing is consistent in that if anything goes wrong, Java throws a runtime exception. That could be a format that doesn't match the `String` to be parsed or an invalid date.

Summary

In this chapter, you learned that `String`s are immutable sequences of characters. The `new` operator is optional. The concatenation operator (`+`) creates a new `String` with the content of the first `String` followed by the content of the second `String`. If either operand involved in the `+` expression is a `String`, concatenation is used; otherwise, addition is used. `String` literals are stored in the string pool. The `String` class has many methods. You need to know `charAt()`, `concat()`, `endsWith()`, `equals()`, `equalsIgnoreCase()`, `indexOf()`, `length()`, `replace()`, `startsWith()`, `substring()`, `toLowerCase()`, `toUpperCase()`, and `trim()`.

`StringBuilder`s are mutable sequences of characters. Most of the methods return a reference to the current object to allow method chaining. The `StringBuilder` class has many methods. You need to know `append()`, `charAt()`, `delete()`, `deleteCharAt()`, `indexOf()`, `insert()`, `length()`, `reverse()`, `substring()`, and `toString()`. `StringBuffer` is the same as `StringBuilder` except that it is thread safe.

Calling `==` on `String` objects will check whether they point to the same object in the pool. Calling `==` on `StringBuilder` references will check whether they are pointing to the same `StringBuilder` object. Calling `equals()` on `String` objects will check whether the sequence of characters is the same. Calling `equals()` on `StringBuilder` objects will check whether they are pointing to the same object rather than looking at the values inside.

An array is a fixed-size area of memory on the heap that has space for primitives or pointers to objects. You specify the size when creating it—for example, `int[] a = new int[6];`. Indexes begin with 0 and elements are referred to using `a[0]`. The `Arrays.sort()` method sorts an array. `Arrays.binarySearch()` searches a sorted array and returns the index of a match. If no match is found, it negates the position where the element would

need to be inserted and subtracts 1. Methods that are passed varargs (...) can be used as if a normal array was passed in. In a multidimensional array, the second-level arrays and beyond can be different sizes.

An `ArrayList` can change size over its life. It can be stored in an `ArrayList` or `List` reference. Generics can specify the type that goes in the `ArrayList`. You need to know the methods `add()`, `clear()`, `contains()`, `equals()`, `isEmpty()`, `remove()`, `set()`, and `size()`. Although an `ArrayList` is not allowed to contain primitives, Java will autobox parameters passed in to the proper wrapper type. `Collections.sort()` sorts an `ArrayList`.

A `LocalDate` contains just a date, a `LocalTime` contains just a time, and a `LocalDateTime` contains both a date and time. All three have private constructors and are created using `LocalDate.now()` or `LocalDate.of()` (or the equivalents for that class). Dates and times can be manipulated using `plus`*`XXX`* or `minus`*`XXX`* methods. The `Period` class represents a number of days, months, or years to add or subtract from a `LocalDate` or `LocalDateTime`. `DateTimeFormatter` is used to output dates and times in the desired format. The date and time classes are all immutable, which means the return value must be used.

Exam Essentials

Be able to determine the output of code using `String`. Know the rules for concatenating `String`s and how to use common `String` methods. Know that `String`s are immutable. Pay special attention to the fact that indexes are zero based and that `substring()` gets the string up until right before the index of the second parameter.

Be able to determine the output of code using `StringBuilder`. Know that `StringBuilder` is mutable and how to use common `StringBuilder` methods. Know that `substring()` does not change the value of a `StringBuilder` whereas `append()`, `delete()`, and `insert()` do change it. Also note that most `StringBuilder` methods return a reference to the current instance of `StringBuilder`.

Understand the difference between `==` and `equals`. `==` checks object equality. `equals()` depends on the implementation of the object it is being called on. For `String`s, `equals()` checks the characters inside of it.

Be able to determine the output of code using arrays. Know how to declare and instantiate one-dimensional and multidimensional arrays. Be able to access each element and know when an index is out of bounds. Recognize correct and incorrect output when searching and sorting.

Be able to determine the output of code using `ArrayList`. Know that `ArrayList` can increase in size. Be able to identify the different ways of declaring and instantiating an `ArrayList`. Identify correct output from `ArrayList` methods, including the impact of autoboxing.

Recognize invalid uses of dates and times. `LocalDate` does not contain time fields and `LocalTime` does not contain date fields. Watch for operations being performed on the wrong time. Also watch for adding or subtracting time and ignoring the result.

Review Questions

1. What is output by the following code? (Choose all that apply)

```
1: public class Fish {
2:  public static void main(String[] args) {
3:    int numFish = 4;
4:    String fishType = "tuna";
5:    String anotherFish = numFish + 1;
6:    System.out.println(anotherFish + " " + fishType);
7:    System.out.println(numFish + " " + 1);
8:  } }
```

   A. `4 1`
   B. `41`
   C. `5`
   D. `5 tuna`
   E. `5tuna`
   F. `51tuna`
   G. The code does not compile.

2. Which of the following are output by this code? (Choose all that apply)

```
3: String s = "Hello";
4: String t = new String(s);
5: if ("Hello".equals(s)) System.out.println("one");
6: if (t == s) System.out.println("two");
7: if (t.equals(s)) System.out.println("three");
8: if ("Hello" == s) System.out.println("four");
9: if ("Hello" == t) System.out.println("five");
```

   A. `one`
   B. `two`
   C. `three`
   D. `four`
   E. `five`
   F. The code does not compile.

3. Which are true statements? (Choose all that apply)
   A. An immutable object can be modified.
   B. An immutable object cannot be modified.
   C. An immutable object can be garbage collected.

D. An immutable object cannot be garbage collected.

E. `String` is immutable.

F. `StringBuffer` is immutable.

G. `StringBuilder` is immutable.

4. What is the result of the following code?

```
7: StringBuilder sb = new StringBuilder();
8: sb.append("aaa").insert(1, "bb").insert(4, "ccc");
9: System.out.println(sb);
```

A. `abbaaccc`

B. `abbaccca`

C. `bbaaaccc`

D. `bbaaccca`

E. An exception is thrown.

F. The code does not compile.

5. What is the result of the following code?

```
2: String s1 = "java";
3: StringBuilder s2 = new StringBuilder("java");
4: if (s1 == s2)
5:  System.out.print("1");
6: if (s1.equals(s2))
7:  System.out.print("2");
```

A. `1`

B. `2`

C. `12`

D. No output is printed.

E. An exception is thrown.

F. The code does not compile.

6. What is the result of the following code?

```
public class Lion {
  public void roar(String roar1, StringBuilder roar2) {
    roar1.concat("!!!");
    roar2.append("!!!");
  }
public static void main(String[] args) {
  String roar1 = "roar";
  StringBuilder roar2 = new StringBuilder("roar");
  new Lion().roar(roar1, roar2);
```

```
  System.out.println(roar1 + " " + roar2);
} }
```

A. `roar roar`

B. `roar roar!!!`

C. `roar!!! roar`

D. `roar!!! roar!!!`

E. An exception is thrown.

F. The code does not compile.

7. Which are the results of the following code? (Choose all that apply)

```
String letters = "abcdef";
System.out.println(letters.length());
System.out.println(letters.charAt(3));
System.out.println(letters.charAt(6));
```

A. `5`

B. `6`

C. `c`

D. `d`

E. An exception is thrown.

F. The code does not compile.

8. Which are the results of the following code? (Choose all that apply)

```
String numbers = "012345678";
System.out.println(numbers.substring(1, 3));
System.out.println(numbers.substring(7, 7));
System.out.println(numbers.substring(7));
```

A. `12`

B. `123`

C. `7`

D. `78`

E. A blank line.

F. An exception is thrown.

G. The code does not compile.

9. What is the result of the following code?

```
3: String s = "purr";
4: s.toUpperCase();
5: s.trim();
6: s.substring(1, 3);
```

```
7: s += " two";
8: System.out.println(s.length());
```

A. 2

B. 4

C. 8

D. 10

E. An exception is thrown.

F. The code does not compile.

10. What is the result of the following code? (Choose all that apply)

```
13: String a = "";
14: a += 2;
15: a += 'c';
16: a += false;
17: if ( a == "2cfalse") System.out.println("==");
18: if ( a.equals("2cfalse")) System.out.println("equals");
```

A. Compile error on line 14.

B. Compile error on line 15.

C. Compile error on line 16.

D. Compile error on another line.

E. ==

F. equals

G. An exception is thrown.

11. What is the result of the following code?

```
4: int total = 0;
5: StringBuilder letters = new StringBuilder("abcdefg");
6: total += letters.substring(1, 2).length();
7: total += letters.substring(6, 6).length();
8: total += letters.substring(6, 5).length();
9: System.out.println(total);
```

A. 1

B. 2

C. 3

D. 7

E. An exception is thrown.

F. The code does not compile.

12. What is the result of the following code? (Choose all that apply)

```
StringBuilder numbers = new StringBuilder("0123456789");
numbers.delete(2,  8);
numbers.append("-").insert(2, "+");
System.out.println(numbers);
```

A. `01+89-`

B. `012+9-`

C. `012+-9`

D. `0123456789`

E. An exception is thrown.

F. The code does not compile.

13. What is the result of the following code?

```
StringBuilder b = "rumble";
b.append(4).deleteCharAt(3).delete(3, b.length() - 1);
System.out.println(b);
```

A. `rum`

B. `rum4`

C. `rumb4`

D. `rumble4`

E. An exception is thrown.

F. The code does not compile.

14. Which of the following can replace line 4 to print `"avaJ"`? (Choose all that apply)

```
3: StringBuilder puzzle = new StringBuilder("Java");
4: // INSERT CODE HERE
5: System.out.println(puzzle);
```

A. `puzzle.reverse();`

B. `puzzle.append("vaJ$").substring(0, 4);`

C. `puzzle.append("vaJ$").delete(0, 3).deleteCharAt(puzzle.length() - 1);`

D. `puzzle.append("vaJ$").delete(0, 3).deleteCharAt(puzzle.length());`

E. None of the above.

15. Which of these array declarations is not legal? (Choose all that apply)

A. `int[][] scores = new int[5][];`

B. `Object[][][] cubbies = new Object[3][0][5];`

C. `String beans[] = new beans[6];`

D. `java.util.Date[] dates[] = new java.util.Date[2][];`

E. `int[][] types = new int[];`

F. `int[][] java = new int[][];`

16. Which of these compile when replacing line 8? (Choose all that apply)

```
7: char[]c = new char[2];
8: // INSERT CODE HERE
```

A. `int length = c.capacity;`
B. `int length = c.capacity();`
C. `int length = c.length;`
D. `int length = c.length();`
E. `int length = c.size;`
F. `int length = c.size();`
G. None of the above.

17. Which of these compile when replacing line 8? (Choose all that apply)

```
7: ArrayList l = new ArrayList();
8: // INSERT CODE HERE
```

A. `int length = l.capacity;`
B. `int length = l.capacity();`
C. `int length = l.length;`
D. `int length = l.length();`
E. `int length = l.size;`
F. `int length = l.size();`
G. None of the above.

18. Which of the following are true? (Choose all that apply)

A. An array has a fixed size.
B. An `ArrayList` has a fixed size.
C. An array allows multiple dimensions.
D. An array is ordered.
E. An `ArrayList` is ordered.
F. An array is immutable.
G. An `ArrayList` is immutable.

19. Which of the following are true? (Choose all that apply)

A. Two arrays with the same content are equal.
B. Two `ArrayLists` with the same content are equal.
C. If you call `remove(0)` using an empty `ArrayList` object, it will compile successfully.
D. If you call `remove(0)` using an empty `ArrayList` object, it will run successfully.
E. None of the above.

20. What is the result of the following statements?

```
6:  List<String> list = new ArrayList<String>();
7:  list.add("one");
8:  list.add("two");
9:  list.add(7);
10: for(String s : list)  System.out.print(s);
```

A. `onetwo`

B. `onetwo7`

C. `onetwo` followed by an exception

D. Compiler error on line 9.

E. Compiler error on line 10.

21. What is the result of the following statements?

```
3: ArrayList<Integer> values = new ArrayList<>();
4: values.add(4);
5: values.add(5);
6: values.set(1, 6);
7: values.remove(0);
8: for (Integer v : values) System.out.print(v);
```

A. `4`

B. `5`

C. `6`

D. `46`

E. `45`

F. An exception is thrown.

G. The code does not compile.

22. What is the result of the following?

```
int[] random = { 6, -4, 12, 0, -10 };
int x = 12;
int y = Arrays.binarySearch(random, x);
System.out.println(y);
```

A. `2`

B. `4`

C. `6`

D. The result is undefined.

E. An exception is thrown.

F. The code does not compile.

23. What is the result of the following?

```
4: List<Integer> list = Arrays.asList(10, 4, -1, 5);
5: Collections.sort(list);
6: Integer array[] = list.toArray(new Integer[4]);
7: System.out.println(array[0]);
```

A. `-1`

B. `10`

C. Compiler error on line 4.

D. Compiler error on line 5.

E. Compiler error on line 6.

F. An exception is thrown.

24. What is the result of the following?

```
6: String [] names = {"Tom", "Dick", "Harry"};
7: List<String> list = names.asList();
8: list.set(0, "Sue");
9: System.out.println(names[0]);
```

A. `Sue`

B. `Tom`

C. Compiler error on line 7.

D. Compiler error on line 8.

E. An exception is thrown.

25. What is the result of the following?

```
List<String> hex = Arrays.asList("30", "8", "3A", "FF");
Collections.sort(hex);
int x = Collections.binarySearch(hex, "8");
int y = Collections.binarySearch(hex, "3A");
int z = Collections.binarySearch(hex, "4F");
System.out.println(x + " " + y + " " + z);
```

A 0 1 –2

B. 0 1 –3

C. 2 1 –2

D. 2 1 –3

E. None of the above.

F. The code doesn't compile.

26. Which of the following are true statements about the following code? (Choose all that apply)

```
4: List<Integer> ages = new ArrayList<>();
5: ages.add(Integer.parseInt("5"));
```

```
6: ages.add(Integer.valueOf("6"));
7: ages.add(7);
8: ages.add(null);
9: for (int age : ages) System.out.print(age);
```

A. The code compiles.

B. The code throws a runtime exception.

C. Exactly one of the add statements uses autoboxing.

D. Exactly two of the add statements use autoboxing.

E. Exactly three of the add statements use autoboxing.

27. What is the result of the following?

```
List<String> one = new ArrayList<String>();
one.add("abc");
List<String> two = new ArrayList<>();
two.add("abc");
if (one == two)
  System.out.println("A");
else if (one.equals(two))
  System.out.println("B");
else
  System.out.println("C");
```

A. `A`

B. `B`

C. `C`

D. An exception is thrown.

E. The code does not compile.

28. Which of the following can be inserted into the blank to create a date of June 21, 2014? (Choose all that apply)

```
import java.time.*;

public class StartOfSummer {

  public static void main(String[] args) {
  LocalDate date = ____________________________
  }

}
```

A. `new LocalDate(2014, 5, 21);`

B. `new LocalDate(2014, 6, 21);`

C. `LocalDate.of(2014, 5, 21);`

D. `LocalDate.of(2014, 6, 21);`

E. `LocalDate.of(2014, Calendar.JUNE, 21);`

F. `LocalDate.of(2014, Month.JUNE, 21);`

29. What is the output of the following code?

```
LocalDate date = LocalDate.parse("2018-04-30", DateTimeFormatter.ISO_LOCAL_
DATE);
date.plusDays(2);
date.plusHours(3);
System.out.println(date.getYear() + " " + date.getMonth() + " "
+ date.getDayOfMonth());
```

A. `2018 APRIL 2`

B. `2018 APRIL 30`

C. `2018 MAY 2`

D. The code does not compile.

E. A runtime exception is thrown.

30. What is the output of the following code?

```
LocalDate date = LocalDate.of(2018, Month.APRIL, 40);
System.out.println(date.getYear() + " " + date.getMonth() + " "
+ date.getDayOfMonth());
```

A. `2018 APRIL 4`

B. `2018 APRIL 30`

C. `2018 MAY 10`

D. Another date.

E. The code does not compile.

F. A runtime exception is thrown.

31. What is the output of the following code?

```
LocalDate date = LocalDate.of(2018, Month.APRIL, 30);
date.plusDays(2);
date.plusYears(3);
System.out.println(date.getYear() + " " + date.getMonth() + " "
+ date.getDayOfMonth());
```

A. `2018 APRIL 2`

B. `2018 APRIL 30`

C. `2018 MAY 2`

D. `2021 APRIL 2`

E. `2021 APRIL 30`

F. `2021 MAY 2`

G. A runtime exception is thrown.

32. What is the output of the following code?

```
LocalDateTime d = LocalDateTime.of(2015, 5, 10, 11, 22, 33);
Period p = Period.of(1, 2, 3);
d = d.minus(p);
DateTimeFormatter f = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(d.format(f));
```

A. `3/7/14 11:22 AM`

B. `5/10/15 11:22 AM`

C. `3/7/14`

D. `5/10/15`

E. `11:22 AM`

F. The code does not compile.

G. A runtime exception is thrown.

33. What is the output of the following code?

```
LocalDateTime d = LocalDateTime.of(2015, 5, 10, 11, 22, 33);
Period p = Period.ofDays(1).ofYears(2);
d = d.minus(p);
DateTimeFormatter f = DateTimeFormatter.ofLocalizedDateTime(FormatStyle
.SHORT);
System.out.println(f.format(d));
```

A. `5/9/13 11:22 AM`

B. `5/10/13 11:22 AM`

C. `5/9/14`

D. `5/10/14`

E. The code does not compile.

F. A runtime exception is thrown.